

## Sequence analysis

# RECKONER: Read Error Corrector Based on KMC

Maciej Długosz<sup>1</sup> and Sebastian Deorowicz<sup>1,\*</sup><sup>1</sup>Institute of Informatics, Silesian University of Technology, Akademicka 16, 44-100 Gliwice, Poland

\*To whom correspondence should be addressed.

Associate Editor: XXXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

## Abstract

**Motivation:** Next-generation sequencing tools have enabled producing of huge amount of genomic information at low cost. Unfortunately, presence of sequencing errors in such data affects quality of downstream analyzes. Accuracy of them can be improved by performing error correction. Because of huge amount of such data correction algorithms have to: be fast, memory-frugal, and provide high accuracy of error detection and elimination for variously-sized organisms.

**Results:** We introduce a new algorithm for genomic data correction, capable of processing eucaryotic 300 Mbp-genome-size, high error-rated data using less than 4 GB of RAM in less than 40 minutes on 16-core CPU. The algorithm allows to correct sequencing data at better or comparable level than competitors. This was achieved by using very robust KMC 2  $k$ -mer counter, new method of erroneous regions correction based on both  $k$ -mer counts and FASTQ quality indicators as well as careful optimization.

**Availability:** Program is freely available at <http://sun.aei.posl.pl/REFRESH>.

**Contact:** [sebastian.deorowicz@polsl.pl](mailto:sebastian.deorowicz@polsl.pl)

## 1 Introduction

For several years we have been witnessing of amazing advances in developing of DNA sequencing technologies. The famous Sanger method (Sanger *et al.*, 1977) has been superseded by so-called next-generation sequencing (NGS) technologies. The instruments by Illumina/Solexa, Roche 454, Ion Torrent, ABI SOLiD (Metzker, 2010) allow producing huge amounts of sequencing reads at low cost. This is, however, occupied by higher error rate. The errors can be classified as: substitutions, which involves altering of nucleotides by erroneous ones and indels, which involves insertions of nucleotide sequences into another sequence or deletions of stretches of nucleotide sequences. In Illumina, currently dominating technology, a major group of errors involve substitutions (Laehnemann *et al.*, 2015).

DNA sequencing data are used in various applications, like *de novo* assembly, reassembly, metagenomics, detecting of single nucleotide polymorphisms (SNPs), gene expression analysis, to enumerate a few. The accuracy of the input data is crucial in all the cases. Therefore, the correction of errors in reads is currently an important and popular issue. The existing solutions are discussed and compared in the recent surveys: (Yang *et al.*, 2013), (Molnar and Ilie, 2015), (Laehnemann *et al.*, 2015). Yang *et al.* classify the correction algorithms into three groups: (i)  $k$ -spectrum-based, (ii) suffix-tree/array-based, and (iii) multiple-sequence-alignment-based.

The algorithms of the first category start from extracting all valid fragments of reads of length  $k$  (called  $k$ -mers). They suppose, that due to data redundancy, the majority of  $k$ -mers would have correspondent

(in sense, that it derives from the same fragment of the genome)  $k$ -mers also placed in other reads. Then, rare  $k$ -mers are altered to the most similar frequent  $k$ -mers. This, of course, means the correction of reads that contain the rare  $k$ -mers. The algorithms tend to introduce the least possible number of changes, or, more precisely, such changes that repair most likely errors. This category includes: Quake (Kelley *et al.*, 2010), RACER (Ilie and Molnar, 2013), BLESS (Heo *et al.*, 2014), Blue (Greenfield *et al.*, 2014), Musket (Liu *et al.*, 2013), Lighter (Li *et al.*, 2014), Trowel (Lim *et al.*, 2014), Pollux (Marinier *et al.*, 2015), BFC (Li, 2015), Ace (Sheikhzadeh and de Ridder, 2015).

The suffix-tree/array-based algorithms, like SHREC (Schröder *et al.*, 2009), HiTEC (Ilie *et al.*, 2011) also extract  $k$ -mers, but they utilize simultaneously different values of  $k$  and store  $k$ -mer sets in a suffix data structure.

The multiple-sequence-alignment-based solutions choose from the input dataset such reads that seem to origin from the same fragment of genome. Then they perform one of multiple sequence alignment algorithms on these reads with aim of finding the consensus form of the reads. This category involves Coral (Salmela and Schröder, 2011), ECHO (Kao *et al.*, 2011), Karect (Allam *et al.*, 2015). Fiona (Schulz *et al.*, 2014) is a hybrid approach, as it uses suffix tree and also performs multiple sequence alignment.

The read correction problem is hard due to the following reasons. The “source” genome is not known in advance, so determining the proper form of a read could be achieved only with heuristic methods. Of course, especially in a case of low coverage, the algorithm can sometimes change the correct symbol to the wrong one. The repeats typical especially for large

genomes, could also cause problems with choosing proper correction from a set of a few possibilities.

The most important feature of the read corrector is of course the quality of the results, but we should remember that the amount of input data is huge, as the number of reads can be counted in hundreds of millions. This can lead to large memory occupation if the algorithm constructs a complex data structure from all the reads. The computation time is also an important factor. The above-described reasons motivate deployment of new solutions in this field.

We present a new algorithm for correction of read errors—RECKONER. Our solution is able to correct eucaryotic 300 Mbp sequencing data using less than 4 GB of RAM in less than 40 minutes on a machine equipped with 16 CPU cores, providing correction accuracy better or comparable to competitive methods. The presented results of evaluation of our and state-of-the-art algorithms are on both real and simulated data, including assay of influence of correction on various statistical indicators of correction quality and on results of typical applications of NGS reads.

RECKONER performs  $k$ -mer counting with an efficient KMC 2 algorithm (Deorowicz *et al.*, 2015) and stores the  $k$ -mer database, required during the correction phase, in a compact data structure provided by KMC API. It also improves previous methods of error detection by more intensive utilization of base quality indicators. RECKONER introduces a new method of rating possible corrections based on both read bases quality indicators and  $k$ -mer counts. The algorithm performs correction in parallel and allows to process the compressed input data.

## 2 Methods

### 2.1 General idea

The general idea of  $k$ -spectrum-based algorithms is similar. First, they perform  $k$ -mer counting, i.e., counting a number of appearances of every substring of length  $k$  present in the input data. The obtained counters are used to determine whether the specified fragment of a read is correct or not. It relies on the observation, that in NGS technologies data redundancy causes, that with reasonably high probability every short fragment of a genome (i.e., with length  $k$ ) would appear multiple times in the input data.

As RECKONER is a  $k$ -spectrum based algorithm, its workflow is in the high level similar to other algorithms from this family. It consists of four stages:

1.  $k$ -mer counting,
2. determining threshold of number of  $k$ -mer appearances, which indicates *trusted* and *untrusted*  $k$ -mers,
3. removing *untrusted*  $k$ -mers from the  $k$ -mer database,
4. correcting the reads.

This solution relies on altering of possibly erroneous bases in order to achieve a read variant which is, with the highest probability, the correct one. The main part of RECKONER is based on BLESS version 0.12, which provided a spine of error correction scheme, although we have complemented it by series of improvements.

The main initial changes are employing KMC 2 (Deorowicz *et al.*, 2015) for  $k$ -mer counting and storing  $k$ -mers together with counters in the KMC database accessible with KMC API. Such solution has allowed us to prepare a new method of rating corrections, which utilizes both read quality indicators and  $k$ -mer counters. Moreover, we have improved the idea of read extension while performing corrections near the read ends. RECKONER uses not only the information of extension success but also about the extension quality. We also improved utilization of base quality indicators. The idea is to use poor quality values to indicate places, which should be checked more accurately while determining proper read form.

RECKONER is strongly time- and memory-optimized and is parallelized with OpenMP.

Meanwhile, BLESS authors released new versions of their software, which were independently supplemented by various novel functions, including  $k$ -mer counting with KMC 2 and algorithm parallelization. Nevertheless, the other improvements introduced in RECKONER are not present in the new release of BLESS.

### 2.2 $k$ -mer counting and cutoff

The first stage of RECKONER is counting the appearances of every  $k$ -mer in all input FASTQ files. It is performed by KMC 2 in non-quality-aware mode. Counting does not distinguish between  $k$ -mers and their reversed compliments, i.e., KMC counts only canonical  $k$ -mers. Canonical  $k$ -mer is defined as lexicographically smaller of a  $k$ -mer and its reverse compliment. This approach reduces memory requirements nearly twice (compared to counting all the  $k$ -mers as they appear in the reads), which is possible since there is no information in the input data about direction of the strand the corrected fragment originates from. Then, an auxiliary tool, called *cutter*, creates  $k$ -mer appearances histogram, i.e., histogram of numbers of  $k$ -mers, which occur in the input data particular number of times. The histogram is used to find the threshold, being a number of  $k$ -mer appearances used to qualify a  $k$ -mer into either group of *trusted* (error-free, in BLESS – *solid*) or *untrusted* (erroneous, *weak*) ones.

The method of determining the threshold is based on the one implemented in BLESS. It exploits the observation, that untrusted  $k$ -mers appear in the input data generally rarely (like one or two times). On the other hand, in NGS technologies trusted  $k$ -mers, providing the adequate input data coverage, have frequencies significantly larger (e.g., ten or more), so one can expect, that there is a region between low-coverage and high-coverage  $k$ -mers with  $k$ -mers with middle-sized frequencies. RECKONER looks for the first minimum in the histogram (i.e., the first value  $x$  such that both the number of  $k$ -mers appearing  $x-1$  times and  $x+1$  times is larger than the number of  $k$ -mers appearing  $x$  times). This value is chosen as the threshold but to prevent from choosing too large number, which can be caused by not unified genome coverage, the threshold is upper-bounded by 5.

Then, KMC database is truncated by removing  $k$ -mers appearing fewer times than the threshold. This reduces the memory consumption during the next stages, as the database of  $k$ -mers is stored in the main memory.

Read correction algorithms differ in how they count the  $k$ -mer counters. Sometimes they use external tools like Jellyfish (Marçais and Kingsford, 2011) in Quake, KMC 2 in RECKONER. The chosen method has great impact on the processing speed and memory consumption of this stage. The algorithms also differ in the data structure used for storing  $k$ -mers (sometimes together with the related counters). BLESS stores such data in a Bloom filter, Quake uses bitmap of size of the entire space of  $k$ -mers. The side effect of these decisions is that both BLESS and Quake cannot utilize the  $k$ -mer counters in the correction phase, as they only know whether the  $k$ -mer is trusted or not. Sometimes even this knowledge is imperfect (e.g., Bloom filters allow for some number of false positive answers of appearance of  $k$ -mers, although this problem has been partially solved in BLESS).

RECKONER uses  $k$ -mer database produced by KMC 2, accessible by KMC API. This way it can relay not only on the presence of  $k$ -mers but also on the exact number of appearances of each  $k$ -mer in the input dataset. KMC 2 allows to calculate the quality-aware counters taking into account base qualities (“ $q$ -mers”) (Kelley *et al.*, 2010). Therefore, we experimented with using them in the early versions of RECKONER, but as they have not bring a significant improvement in the quality of a correction, we finally discarded this approach.

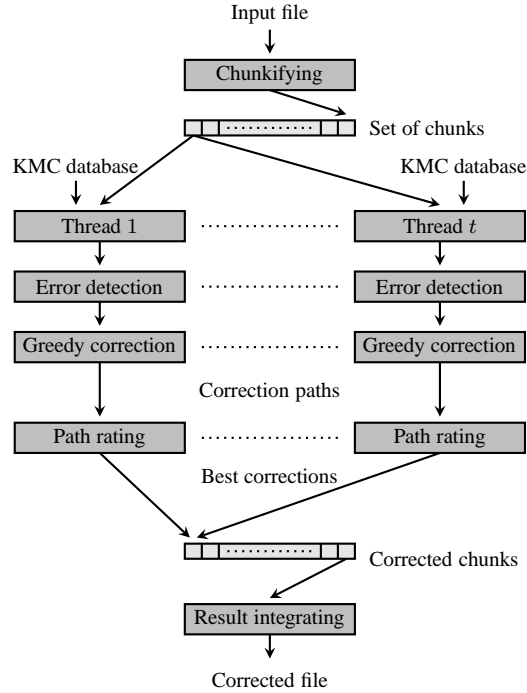


Fig. 1. Correction process (first  $k$ -mer correction is disregarded)

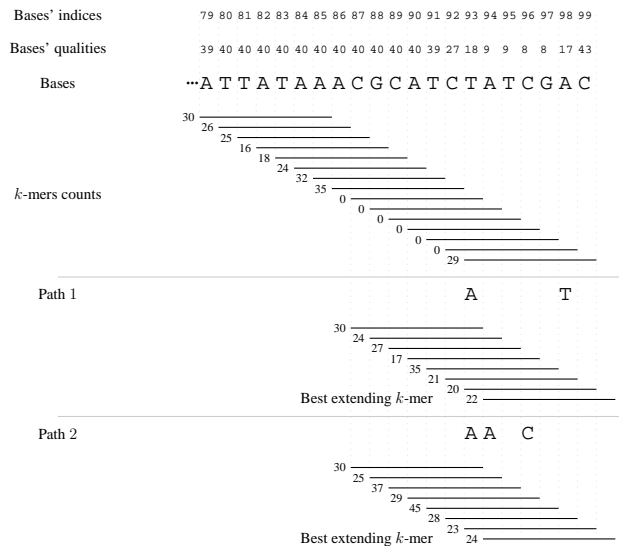


Fig. 2. Example of correction paths rating

## 2.3 Correcting of the reads

### Choice of erroneous read regions

The last stage of the processing, the read correction, consists of a few steps applied separately for every FASTQ file (Fig. 1): (i) read scanning and partitioning (*chunkifying*) for multithreading, (ii) detecting of the erroneous regions, (iii) generating and rating of error correction paths, and (iv) results integrating.

The idea of read scanning and its parameters are similar to corresponding step of BLESS. It relies on reading every input file with aim of dividing it into parts (*chunks*) processed separately by different threads.

Every correction thread processes a subset of chunks appointed in the previous step. First, every read is split into regions that are marked as either proper or erroneous. Erroneous regions are determined by extracting its  $k$ -mers and by checking their presence in the KMC database. If a current  $k$ -mer is absent from the database, it makes a suggestion, that it originates from a read region having at least one erroneous base. Erroneous regions are expanded if the adjacent bases have low qualities (i.e., smaller than 10). Moreover, both type regions can be removed if their lengths are too small (less than 2).

Fig. 2 shows an example of the process of finding erroneous regions. RECKONER extracts a set of  $k$ -mers from an exemplary read and obtains their counters from the KMC database. The last  $k$ -mer has counter 29 despite of presence of low-quality bases within. Next six  $k$ -mers (from right to left) have number of appearances 0 (they have been removed from the database). The presence of such  $k$ -mers causes that RECKONER marks as an erroneous region of indices from 93 to 98. Then, it expands this region to base 99, to remove "proper" region with length shorter than 2 bases.

### Error correction and correction rating

Next step consists of introducing changes into reads (Fig. 2). The method of correction of detected wrong bases exploits greedy algorithm similar to competitive solutions, like BLESS and Quake. The essential part of the algorithm is described below. Every base in an erroneous region, respectively from right to left (i.e., 3' to 5' end – situation 1), for the erroneous region situated in the 5' end, or from left to right (i.e., 5' to 3' end – situation 2), for the other regions, can be altered to another one until the adequate  $k$ -mer with the base become trusted; low-quality bases are changed even if the  $k$ -mer is already trusted. Especially, situation 2 includes erroneous regions situated between two proper regions.

Following its definition, it can be expected, that the erroneous region is of length at least  $k$ . If it is shorter or there is no proper region in the read, then correction of the first  $k$ -mer is performed and after successful result of this correction the standard correction in situation 2 is performed. The correction of the first  $k$ -mer relies on altering bases with quality indicators smaller than 10 or, in case of failure, by altering every single base in the  $k$ -mer.

Every base present in the erroneous region is a candidate to be modified, however, modifications are introduced only to bases satisfying at least one of two conditions: (i) the base makes the trailing (situated contrariwise to the correction direction)  $k$ -mer untrusted or (ii) it has a low value of its quality indicator (accompanying the base in the read).

The second condition is introduced in RECKONER. The idea is to prevent a situation when some read region contains a large number of low-quality bases, strongly suggesting that some of them are erroneous, even if the  $k$ -mers extracted from the region are trusted.

Finally, many candidate solutions can be obtained. The next step is paths rating to choose the best one.

The exemplary read would be corrected by RECKONER starting from base 93. RECKONER would try to change the base unless  $k$ -mer starting at 87th position will become trusted. Let us suppose, that the only possibility is to change it to A. Then, it checks whether  $k$ -mer starting at 88th position become trusted. If yes, it continues correcting following bases until region's  $k$ -mers become trusted. Let us suppose, that the only possibility is to change the 98th base to T. These two changes constitute the first path of changes.

When RECKONER detects a base with low quality (smaller than 10), which has not to be changed to cause a current  $k$ -mer become trusted, it treats "no change" possibility as the beginning of one correction path, and regardless of trust of the  $k$ -mer tries to change the base; if the change causes, that  $k$ -mer is still trusted, then the change starts the next correction path. Let us suppose, that base 94 can be changed to A to cause (regardless

of being trusted after changing base 93), that  $k$ -mer starting from the 88th position remains trusted. This way we have started the second correction path. Let us suppose, that to cause the entire erroneous region to be trusted, we also have to change base 96 to C.

Like BLESS, RECKONER performs read extending. If a corrected base is located nearly ends of the read, there is no possibility to extract sufficient number of  $k$ -mers the base builds. The solution relies on checking also symbols, that possibly would be a continuity of the read in aim of finding such combination of that symbols, that could constitute trusted  $k$ -mers with symbols on the specified end of the read. If for a specified correction path there is no possibility to find extending symbols then the current correction the path is rejected.

Correction of bases with greedy algorithm would cause in rare situations extreme increase of a number of considered paths of correction. Because of that, we have introduced limitations on the maximum number of changes performed in a considered region of a read. If the limit is reached, searching of paths for the region is interrupted.

In case of finding multiple paths of correction there is a need to rank them. This rating is based on the following formula. Let us define read  $r$  as a sequence over the alphabet  $\Sigma = \{A, C, G, T\}$  of length  $\ell$ . If the sequencing data contains another symbols, then they are altered to A. Let  $r[s]$  denotes the  $s$ -th symbol of  $r$ ,  $-1 \leq s \leq \ell$ , where  $r[0]r[1] \dots r[\ell-1]$  are symbols originating from the input read;  $r[-1]$  and  $r[\ell]$  are symbols belonging to the read extension (if it includes that symbols). Moreover,  $r[a, b]$  means  $r[a]r[a+1] \dots r[b]$ .

Let us define  $p$  as a sequence of length  $\ell$  of probabilities, that the particular symbols of the corresponding sequence  $r$  are incorrect. The notation  $p[t]$  means the  $t$ -th value of  $p$ ,  $0 \leq t \leq \ell - 1$ .

Let us suppose, that  $r[a, b]$  is an erroneous region of  $r$ , where  $0 \leq a < b \leq \ell - 1$ . Moreover, in situation 1 we know that  $b < \ell - k + 1$  and in situation 2 that  $a \geq k - 1$ . Let us denote as  $m$  index of the left-most (1) or the right-most (2) modified base. The distance  $d$  of  $m$  from the read's end is:

$$d = \begin{cases} m & (1), \\ \ell - m - 1 & (2). \end{cases}$$

The number  $e$  of bases extending the read is determined as:

$$e = \begin{cases} 0 & \text{if } d \geq k, \\ \min(k - d - 1, 5) & \text{otherwise.} \end{cases}$$

By set  $K_{\text{cov}}$  of  $k$ -mers covering a region  $r[a, b]$  we mean:

$$K_{\text{cov}} = \begin{cases} \{r[a, a+k-1], \dots, r[b, b-k+1]\} & (1), \\ \{r[a-k+1, a], \dots, r[b+k-1, b]\} & (2). \end{cases}$$

The first extending  $k$ -mer  $x_{\text{ext}}$  is defined as:

$$x_{\text{ext}} = \begin{cases} r^*[-1, k-2] & (1), \\ r^*[ \ell - k + 1, \ell ] & (2). \end{cases}$$

and, finally, the set  $K^*$  of rating  $k$ -mers is defined as:

$$K^* = \begin{cases} K_{\text{cov}}^* & \text{if } e = 0, \\ K_{\text{cov}}^* \cup x_{\text{ext}} & \text{if } e > 0. \end{cases}$$

where  $K_{\text{cov}}^*$  is an equivalent of  $K_{\text{cov}}$  containing  $k$ -mers taken from a modified read  $r^*$ , i.e., the version of  $r$  with applied modifications according to the currently evaluated correction path.

If there are more than one possible (trusted) first extending  $k$ -mer,  $K^*$  contains the one with the biggest number of appearances (in a tie we choose any of them, since here only the  $k$ -mer counter is important).

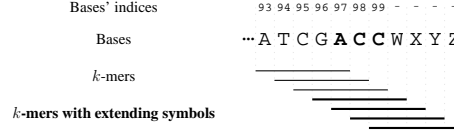


Fig. 3. 5-mers covering bases 97 – 99; W, X, Y, Z – any extending symbols (at most 5)

For rating we use only the first extending  $k$ -mer, however, the necessary condition for acceptance and rating of a path is that every  $x \in K^*$ , is trusted and at least one  $k$ -mer built with every of extending (at most 5) positions is trusted (Fig. 3).

Let  $\text{count}(x)$  be the number of  $k$ -mer  $x$  appearances and weight be defined as follows:

$$\text{weight}(x) = \begin{cases} 1 & \text{if } x \text{ contains no extending bases,} \\ 0.5 & \text{if one base of } x \text{ is the extending base.} \end{cases}$$

We also need prob function defined as follows:

$$\text{prob}(i) = \begin{cases} p[i] & \text{if the } i\text{-th symbol has been changed,} \\ 1 & \text{otherwise.} \end{cases}$$

Then, the region's correction rate is:

$$\text{rate}(a, b, K^*) = \frac{\left( \sum_{x \in K^*} \text{weight}(x) \text{count}(x) \right) \left( \prod_{i=a}^b \text{prob}(i) \right)}{\sum_{x \in K^*} \text{weight}(x)}.$$

Finally, the correction with the biggest rate is applied to the erroneous region.

The base qualities and finally base error probability, can be obtained from the input FASTQ file and the number of  $k$ -mer appearances are taken from the KMC database. The idea of using base probabilities for rating of correction paths has been previously proposed in Quake (Kelley *et al.*, 2010), but there the probabilities have been processed as a parameters of a nonparametric regression. The advantage of the proposed method is utilizing both  $k$ -mer counts and read base qualities for corrections rating. Particular erroneous regions are rated separately.

Table 1. Bases qualities with corresponding probabilities.

Quality	Probability	Quality	Probability
18	0.015849	8	0.158489
9	0.125893	17	0.019953

Based on Table 1, RECKONER would determine the rates for the exemplary paths 1 and 2 as:

$$\begin{aligned} \text{rate}_1 &= \frac{(30 + 24 + 27 + 17 + 35 + 21 + 20 + 0.5 \cdot 22)}{1 + 1 + 1 + 1 + 1 + 1 + 1 + 0.5} \cdot (0.015849 \cdot 0.019953) = 0.0078 \\ \text{rate}_2 &= \frac{(30 + 25 + 37 + 29 + 45 + 28 + 23 + 0.5 \cdot 24)}{1 + 1 + 1 + 1 + 1 + 1 + 1 + 0.5} \cdot (0.015849 \cdot 0.125893 \cdot 0.158489) = 0.009656 \end{aligned}$$

so in the shown situation RECKONER would apply path 2 to the input read.

Table 2. Real datasets used in the experiments

Organism	Accession no.	Genome len.	No. of reads	Read length
<i>S. cerevisiae</i>	ERR422544	12.3Mbp	4.78M	100bp
<i>C. elegans</i>	SRR543736	102.3Mbp	57.72M	101bp
<i>M. acuminata</i>	ERR204808	472.2Mbp	67,18M	108bp

### 3 Results

We have performed a series of tests on both simulated and real data for various-sized genomes, different sequencing coverages, read lengths, and data qualities. These allowed us to compare the examined algorithms from different points of view: correction quality characterized respectively by statistical measures (simulated data), impact on *de novo* assembly and reassembly (real data), memory requirements, and time consumption. For testing we picked data from different Illumina sequencers.

We compared RECKONER and the following state-of-the-art algorithms: Ace (Sheikhzadeh and de Ridder, 2015), BFC (Li, 2015), BLESS (Heo *et al.*, 2014), Blue (Greenfield *et al.*, 2014), Karect (Allam *et al.*, 2015), Lighter (Li *et al.*, 2014), Musket (Liu *et al.*, 2013), Pollux (Marinier *et al.*, 2015), RACER (Ilie and Molnar, 2013), Trowel (Lim *et al.*, 2014). We excluded from this group the popular Quake due to its poor results according to the recent studies (Heo *et al.*, 2014; Lim *et al.*, 2014; Liu *et al.*, 2013; Marinier *et al.*, 2015).

The experiments were performed on a computer equipped with 128 GB of RAM, four AMD Opteron 8378 processors, running under Fedora 16 x86-64 OS. RECKONER was written in C++ and parallelized using OpenMP 3.1 library. For compilation we used G++ 4.7.

#### 3.1 Real data evaluation

In practice, the sequenced reads are mapped (reassembled) or *de novo* assembled. We performed an evaluation of correction algorithms on real data by assaying an impact of the correction on results in both applications as they are sensitive to sequencing errors presence. The tests were performed for three datasets described in Table 2. Two of this datasets were taken from the survey by Molnar and Ilie (2015).

##### De novo assembly

*De novo* assembly is a process of building contiguous sequences of nucleotides, called *contigs*, typically by detection of overlapping fragments of reads (Kelley *et al.*, 2010).

In genome assembly finding of false overlaps or missing true overlaps can occur due to sequencing errors. It may cause generation of false contigs, emergence of ambiguity or breaking of true contigs. Moreover, many of assemblers model contig fragments and overlaps between them with a de Bruijn graph. De Bruijn graph memory requirements strongly depend on a number of distinct *k*-mers in the input data. Every additional (e.g., erroneous) *k*-mer causes at most *k* faulty nodes appearing in the graph, what significantly impacts on memory requirement of assembly.

The quality of assembly was evaluated using several measures, i.e., NG50, NA50, N50. In the main text the NG50 measure (such contig length, that contigs of length NG50 or more consist 50% of the full genome) is used. The other results (together with time of assembly and assembly memory requirements are given in Supplementary material). For experiments we used Velvet assembler (Zerbino and Birney, 2008). NG50 values were determined with Quast (Gurevich *et al.*, 2013).

The results are given in Table 3. As it can be seen the best results are obtained by Karect, but the second place is for RECKONER. What is important, only Karect, RECKONER, and Blue are always ranked among

Table 3. Results of de novo assembly

Corrector	Organism		
	<i>S. cerevisiae</i>	<i>C. elegans</i>	<i>M. acuminata</i>
Karect	<b>18,034</b>	2,874	<b>1,469</b>
RECKONER	17,800	<b>2,973</b>	1,384
Blue	16,991	2,952	1,423
Pollux	16,304	2,900	1,423
BFC	17,292	2,731	1,342
BLESS	16,861	2,783	1,332
Lighter	16,167	2,848	1,318
Musket	16,639	2,562	1,279
RACER	17,292	2,040	1,092
<i>Without corr.</i>	16,204	2,521	1,335
Ace	16,574	1,294	913
Trowel	13,831	1,626	1,261

The given numbers are NG50 values obtained by Velvet assembler for data corrected using the examined correctors. The correctors are ordered according to average rank. The best values are in bold

Table 4. Results of mapping

Corrector	Organism		
	<i>S. cerevisiae</i>	<i>C. elegans</i>	<i>M. acuminata</i>
Karect	93.83	<b>81.99</b>	<b>82.70</b>
RECKONER	93.76	81.58	82.46
RACER	<b>93.99</b>	81.79	82.04
BLESS	93.78	81.56	82.30
Blue	93.77	81.70	82.21
BFC	93.76	81.67	82.05
Musket	93.69	81.54	82.31
Ace	93.78	81.52	82.10
Lighter	93.73	81.52	82.30
Trowel	92.92	81.56	82.26
Pollux	93.64	81.42	82.45
<i>Without corr.</i>	93.65	81.45	82.04

The given numbers are fractions (expressed in %) of mapped reads. The correctors are ordered according to average rank. The best values are in bold

first 5 positions. The other algorithms perform poorly for at least one dataset. Two of them perform even poorer than when no correction is used.

##### Reassembly

The goal of reassembly is to detect reads' locations in the genome by aligning them to the reference genome. It cannot be done by simple searching of each read in the reference genome. Intraspecific diversity, especially dissimilarities of single nucleotides between genomes of the same species (single nucleotide polymorphisms), enforce use of more sophisticated algorithms. In this case sequencing errors cause ambiguities of reads matching and finding false matchings or missing true matchings.

We performed the evaluation using Bowtie 2 (Langmead and Salzberg, 2012). As a measurement indicator we picked both number of modifications required to be introduced to align a read and number of reads successfully aligned to the genome. Table 4 presents the fractions of reads mapped by Bowtie 2 after correction. Other results, i.e., time of mapping, memory consumption of Bowtie 2, fractions of reads mapped uniquely, fractions of reads mapped with 1, 2, ..., 5 and more mismatches are given in Supplementary material. The results show that Karect allows to map the largest number of reads, following by RECKONER, RACER, BLESS, and Blue. Nevertheless, the absolute values are very close, especially for *S. cerevisiae* dataset.

Table 5. Results of correction using simulated reads

Corrector	Sensitivity	Precision	Gain	RAM [GB]	Time [s]	Corrector	Sensitivity	Precision	Gain	RAM [GB]	Time [s]
<b><i>S. cerevisiae</i>, coverage 30×</b>											
read length 100bp, $p = 2.0\%$						read length 100bp, $p = 5.5\%$					
RECKONER	<b>99.077</b>	99.972	<b>99.049</b>	0.536	83	RECKONER	<b>95.718</b>	99.974	<b>95.693</b>	0.573	88
BLESS	98.069	99.991	98.060	0.835	119	BLESS	93.525	99.990	93.516	0.927	109
BFC	77.030	99.996	77.026	1.148	83	BFC	68.406	99.997	68.404	1.157	<b>87</b>
Lighter	80.470	99.923	80.407	<b>0.065</b>	94	Lighter	70.303	99.930	70.254	<b>0.067</b>	97
RACER	80.819	99.946	80.775	1.827	184	Trowel	71.648	99.975	71.631	6.674	140
Musket	60.356	<b>100.000</b>	60.356	0.303	187	RACER	70.766	99.950	70.731	1.827	179
Trowel	74.831	99.990	74.823	4.205	<b>64</b>	Musket	51.886	<b>100.000</b>	51.886	0.304	245
Karect	94.608	99.932	94.544	7.005	168	Ace	91.672	99.921	91.600	2.701	2109
Ace	96.555	99.904	96.462	2.164	1890	Karect	85.665	99.940	85.614	7.005	472
Blue	72.195	99.316	71.698	1.855	582	Blue	61.544	99.519	61.246	2.872	640
Pollux	42.993	90.728	38.599	4.013	949	Pollux	36.297	91.789	33.050	4.558	1060
<b><i>C. elegans</i>, coverage 20×</b>											
read length 150bp, $p = 1.9\%$						read length 151bp, $p = 4.2\%$					
BLESS	95.776	99.997	95.774	1.993	579	RECKONER	<b>96.757</b>	99.992	<b>96.749</b>	1.619	536
RECKONER	<b>98.137</b>	99.968	<b>98.105</b>	1.864	518	BLESS	94.240	<b>100.000</b>	94.240	1.940	697
Lighter	82.244	99.994	82.239	<b>0.438</b>	508	Lighter	63.117	99.994	63.113	<b>0.438</b>	498
BFC	82.160	<b>99.999</b>	82.159	3.702	409	BFC	61.761	99.999	61.760	3.705	501
Musket	74.155	99.999	74.154	0.950	1294	Trowel	76.982	99.778	76.810	15.364	<b>342</b>
Ace	89.606	99.995	89.601	13.172	12787	Musket	37.152	99.998	37.151	0.963	1915
Karect	93.654	99.978	93.634	39.443	4265	Karect	96.203	99.985	96.188	39.471	3026
RACER	76.433	99.987	76.424	13.619	1047	Ace	85.085	99.994	85.080	17.276	14389
Trowel	67.257	99.838	67.148	15.737	<b>143</b>	RACER	33.529	99.983	33.523	13.618	1189
Blue	42.958	97.513	41.863	11.543	5315	Blue	43.745	98.670	43.156	11.702	7612
Pollux	41.097	75.587	27.823	15.262	22428	Pollux	30.039	80.880	22.938	30.657	23445
<b><i>M. acuminata</i>, coverage 30×</b>											
read length 150bp, $p = 1.9\%$						read length 151bp, $p = 4.2\%$					
BLESS	<b>93.574</b>	99.573	<b>93.172</b>	3.717	3051	RECKONER	<b>92.820</b>	99.198	<b>92.070</b>	3.725	2285
BFC	79.321	99.873	79.221	13.186	<b>1622</b>	BLESS	88.800	99.758	88.584	3.727	3310
RECKONER	93.328	98.616	92.018	3.723	2021	BFC	58.549	99.910	58.496	13.184	1835
Lighter	78.215	98.964	77.396	<b>1.336</b>	1977	Lighter	57.952	99.157	57.460	<b>1.336</b>	2023
Musket	71.614	<b>100.000</b>	71.614	3.328	5175	Blue	68.392	98.642	67.451	37.104	<b>1380</b>
Karect	88.183	97.487	85.910	103.137	5627	Musket	35.366	<b>99.999</b>	35.366	3.386	8627
Blue	67.139	97.469	65.395	35.010	1669	Karect	89.751	98.502	88.387	103.421	12352
RACER	72.467	97.436	70.560	45.066	5056	Trowel	46.441	97.716	45.355	57.607	2440
Trowel	48.584	96.107	46.616	57.547	1936	RACER	40.113	97.305	39.002	45.060	4754
Pollux	out of time (> 12 hours)					Pollux	out of time (> 12 hours)				
Ace	out of time (> 12 hours)					Ace	out of time (> 12 hours)				

The values  $p$  are average probabilities of base error. The correctors are ordered according to average rank. The best values are in bold

### 3.2 Simulated data evaluation

The most direct method of evaluation of correction algorithms is performing tests on reads generated *in silico*, so we simulated genome sequencing by generating a set of ideal reads and introducing errors to them.

The applied method of reads generation was proposed and used in (Kelley *et al.*, 2010) and (Liu *et al.*, 2013). First of all, we needed to choose the expected coverage and read length. Then, we excerpted fragments of the specified length from a full reference genome. To introduce changes imitating sequencing errors we took real quality indicators from FASTQ files with reads of length of reads being generated. Details of datasets being a source of probabilities—patterns—are given in the Supplementary Table 2. Patterns have been selected to represent reads of lengths about 100bp and 150bp and two levels of data quality. (The reported base error probability was calculated as an average among the whole FASTQ file.) Generation has been performed for three differently sized organisms:

*S. cerevisiae*, *C. elegans*, and *M. acuminata* for different read coverages (20× and 30×).

The parameters of algorithms, especially  $k$ -mer lengths, have been determined empirically by choosing the best value for a specified algorithm (according to the preliminary experiments; results not shown).

Each corrected read was assigned to one of the following categories:

- TP (true positive)—before correction the read contained errors and it was perfectly corrected,
- TN (true negative)—before correction the read contained no errors and it remained error-free after correction,
- FP (false positive)—before correction the read contained no errors, but the algorithm introduced at least one error to it,
- FN (false negative)—before correction the read contained errors, but the algorithm was not able to correct it properly, i.e., errors were not corrected, were miscorrected, or the algorithm introduced new errors.

For comparison of correction accuracy we used three statistical measurements: sensitivity =  $|TP|/(|TP| + |FN|)$ , precision =  $|TP|/(|TP| + |FP|)$ , gain =  $|TP| - |FP|/(|TP| + |FN|)$ .

The selection of results is given in Table 5. Full results and plots are given in Supplementary material. Results show that RECKONER is in all cases in a group of the three best algorithms and in four cases it is the best one, what is the effect of the highest values of gain and sensitivity in most cases and values of time and memory being close to best.

RECKONER achieves best results for data of poorer quality. In all cases RECKONER, BLESS, BFC, and Lighter are ranked in the top 4 places.

The important observation is Karect's memory requirements. Karect allows to specify the upper limit of memory consumption (we limited it to 120 GB), but choosing values acceptable for a typical PC (e.g., 16 or 32 GB) causes considerable increase of computational time and moderate decrease of quality.

## 4 Conclusion

We have presented RECKONER, an efficient error corrector for the sequencing data. It is based on the BLESS code. Nevertheless, RECKONER implements several new ideas that allowed us to obtain highly competitive results when compared to the state-of-the-art algorithms. In simulated-data experiments it was usually the best according to the gain and sensitivity measures. RECKONER is also among the fastest and most memory frugal algorithms that allows to run it even on a commodity personal computer for quite large data.

In the real-data experiments, in which we evaluated both the quality of mapping of corrected reads and quality of *de novo* assembly, it was at the second place, just after Karect. Nevertheless, the memory requirements of Karect are much larger, which limits its applications to rather powerful computing servers.

## Funding

This work was supported by the Polish National Science Centre under the project DEC-2012/05/B/ST6/03148. The work was performed using the infrastructure supported by POIG.02.03.01-24-099/13 grant: "GeCONiL-Upper Silesian Center for Computational Science and Engineering".

## References

- Allam,A., Kalnis,P., Solovyev,V. (2015) Karect: Accurate Correction of Substitution, Insertion and Deletion Errors for Next-generation Sequencing Data, *Bioinformatics*, **31**(21), 3421–3428.
- Deorowicz,S., Debudaj-Grabysz,A., Grabowski,S. (2013) Disk-based *k*-mer counting on a PC, *BMC Bioinformatics*, **16**, Article no. 160.
- Deorowicz,S., Kokot,M., Grabowski,S., Debudaj-Grabysz,A. (2015) KMC 2: Fast and resource-frugal *k*-mer counting, *Bioinformatics*, **31**(10), 1569–1576.
- Greenfield,P., Duesing,K., Papanicolaou,A., Bauer,D. (2014) Blue: correcting sequencing errors using consensus and context, *Bioinformatics*, **30**(19), 2723–2732.
- Gurevich,A., Saveliev,V., Vyahhi,N., Tesler,G. (2013) QUAST: quality assessment tool for genome assemblies, *Bioinformatics*, **29**(8), 1072–1075.
- Heo,Y., Wu,X.-L., Chen,D., Ma,J., Hwu,W.-M. (2014) BLESS: Bloom-filter-based Error Correction Solution for Highthroughput Sequencing Reads, *Bioinformatics*, **30**(10), 1354–1362.
- Ilie,L., Fazayeli,F., Ilie,S. (2011) HiTEC: accurate error correction in high-throughput sequencing data, *Bioinformatics*, **27**(3), 295–302.
- Ilie,L., Molnar,M. (2013) RACER: Rapid and Accurate Correction of Errors in Reads, *Bioinformatics*, **29**(19), 2490–2493.
- Kao,W.-C., Chan,A., Song,Y. (2011) ECHO: a reference-free short-read error correction algorithm, *Genome Res*, **21**, 1181–1192.
- Kelley,D., Schatz,M., Salzberg,S.L. (2010) Quake: quality-aware detection and correction of sequencing errors, *Genome Biol*, **11**:R116.
- Laehnmann,D., Borkhardt,A., McHardy,A.C. (2015) Denoising DNA deep sequencing data – high-throughput sequencing errors and their correction, *Brief Bioinform*, **16**, 588–599.
- Langmead,B., Salzberg,S.L. (2012) Fast gapped-read alignment with Bowtie 2, *Nat Methods*, **9**, 357–359.
- Li,H. (2015) BFC: correcting Illumina sequencing errors, *Bioinformatics*, **31**(17), 2885–2887.
- Li,S., Florea,L., Langmead,B. (2014) Lighter: fast and memory-efficient sequencing error correction without counting, *Genome Biol*, **15**:509.
- Lim,E.-C., Müller,J., Hagmann,J., Henz,S., Kim,S.-T., Weigel,D. (2014) Trowel: a fast and accurate error correction module for Illumina sequencing reads, *Bioinformatics*, **30**(22), 3264–3265.
- Liu,Y., Schröder,J., Schmidt,B. (2013) Musket: a multistage *k*-mer spectrum-based error corrector for Illumina sequence data, *Journal Name, Bioinformatics*, **29**(3), 308–315.
- Marçais,G., Kingsford,C. (2011) A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers, *Bioinformatics*, **27**(6), 764–770.
- Marinier,E., Brown,D.G., McConkey,B.J. (2015) Pollux: platform independent error correction of single and mixed genomes, *BMC Bioinformatics*, **16**, Article no. 435.
- Metzker,M.L. (2010) Sequencing technologies – the next generation, *Nat Rev Genet*, **11**(1), 31–46.
- Molnar,M., Ilie,L. (2015) Correcting Illumina data, *Brief Bioinform*, **16**(4), 588–599.
- Salmela,L., Schröder,J. (2011) Correcting errors in short reads by multiple alignments, *Bioinformatics*, **27**(11), 1455–1461.
- Sanger,F., Nicklen,S., Coulson,A.R. (1977) DNA sequencing with chain-terminating inhibitors, *Proc Natl Acad Sci USA*, **74**(12), 5463–5467.
- Schröder,J., Schröder,H., Puglisi,S., Sinha,R., Schmidt,B. (2009) SHREC: a short-read error correction method, *Bioinformatics*, **25**(17), 2157–2163.
- Schulz,M.H. *et al.* (2014) Fiona: a parallel and automatic strategy for read error correction, *Bioinformatics*, **30**(17), i356–i363.
- Sheikhzadeh,S., de Ridder,D. (2015) ACE: Accurate Correction of Errors using *K*-mer tries, *Bioinformatics*, **31**(19) 3216–3218.
- Yang,X., Chockalingam,S.P., Aluru,S. (2013) A survey of error-correction methods for next-generation sequencing, *Brief Bioinform*, **14**(1), 56–66.
- Zerbino,D.R., Birney,E. (2008) Velvet: Algorithms for *de novo* short read assembly using de Bruijn graphs, *Genome Res*, **18**, 821–829.

# RECKONER: Read Error Corrector Based on KMC

## Supplementary material

Maciej Długosz

Sebastian Deorowicz

## 1 Algorithms' parameters

The test was performed on correction algorithms' versions shown in Table 1.

Algorithm	Version
RECKONER	0.1
RACER	1.0.1
BLESS	0.24
Lighter	1.0.4
Musket	1.1
Blue	1.1.2
Trowel	0.1.4.3
BFC	BFC-ht, version v1
Pollux	1.00
Ace	1.01
Karect	1.0

Table 1: Correctors versions used in the tests.

The following commands was used for running the algorithms. Algorithms parameters was, i.a.:

- **INPUT** – input FASTQ file,
- **OUTPUT** – output FASTQ file,
- **K** –  $k$ -mer length,
- **GENOME\_LENGTH** – estimated genome length,
- **COVERAGE** – average sequencing coverage.

### RECKONER

```
./run.sh K tmp 16 INPUT
```

### RACER

```
./RACER_Linux_parallel INPUT OUTPUT GENOMELENGTH
```

### BLESS

```
./bless -read INPUT -prefix OUTPUT -kmerlength K
```



### Lighter

Lighter requires parameter  $\alpha$  (ALPHA), which was computed by a formula proposed in the preprint:  $\alpha = 0.05 \frac{70}{\text{COVERAGE}}$ .

```
./lighter -r INPUT -k K GENOMELENGTH ALPHA -t 16
```

### Musket

```
./musket -k K KMERS -p 16 -o OUTPUT INPUT -inorder
```

### Blue

As Blue is a software made in C# and we tested the algorithms on Linux, we used Mono to run it. CUTOFF parameter was computed with a method utilized in BLESS and RECKONER.

```
mono ./Tessel.exe -k K -g GENOMELENGTH -t 16 \
-f fastq -tmp tmp INPUT_cbt INPUT.fastq
mono ./GenerateMerPairs.exe -t 16 INPUT_cbt_K.cbt INPUT
mono ./Blue.exe -m CUTOFF -t 16 -r o -o tmp -f fastq INPUT_cbt_K.cbt INPUT.fastq
```

### Trowel

The file `fastq_files` contained the INPUT path.

```
./trowel -k K -t 16 -f fastq_files -ntr
```

### BFC

```
./bfc -s GENOMELENGTH -t16 > OUTPUT
```

### Pollux

```
./pollux -k K -i INPUT -s true -n false -d false -f false
```

### Ace

```
./ace GENOMELENGTH INPUT OUTPUT
```

### Karect

```
./karect -correct -threads=16 -matchtype=hamming \
-celltype=diploid -inputfile=INPUT -memory=120
```

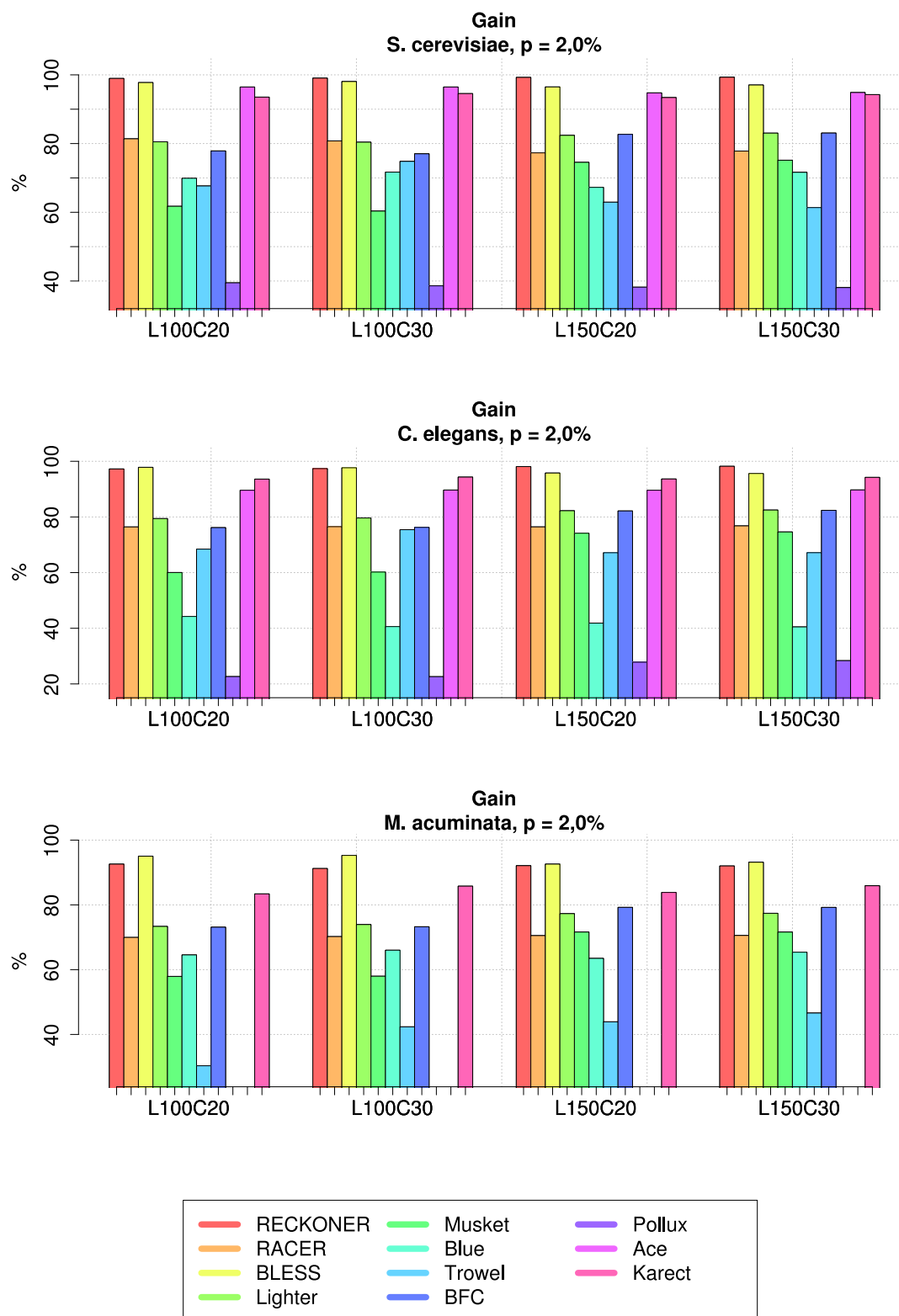
## 2 Results of simulated data correction

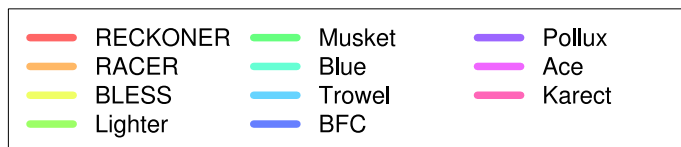
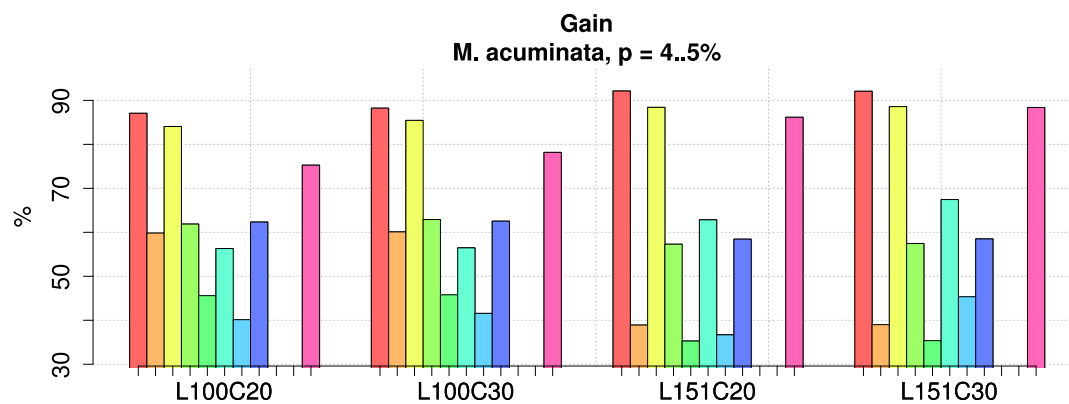
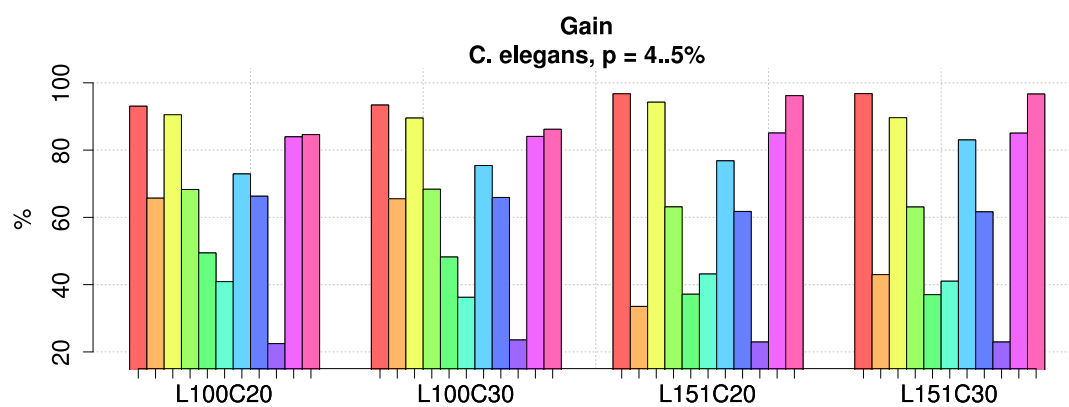
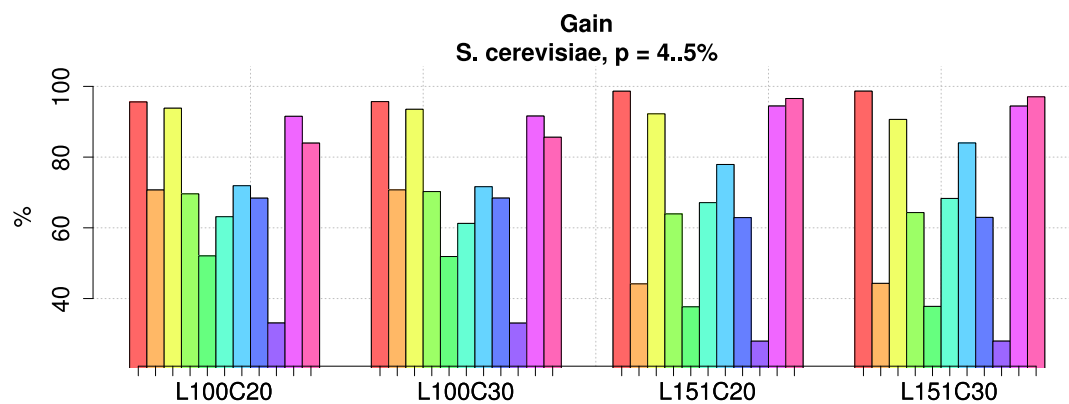
The simulated datasets was generated for three different-sized organisms (*Saccharomyces cerevisiae*, *Caenorhabditis elegans*, and *Musa acuminata*), for two read lengths (100 bp and ca. 150 bp), for two coverages (20 and 30) and two levels of quality (ca. 2% and 4...5% errors, shown in Table 2 with patterns being source of the quality indicators for generation). The following plots show obtained results in terms of gain, sensitivity and precision values and requirements for memory space and computational time. The notation  $LxCy$  means the read length equals to  $x$  bp and the coverage equals to  $y$ ;  $p$  denotes the average probability of a base error. Missing bars denote failures of the correction caused by too huge time consumption (> 12 hours).

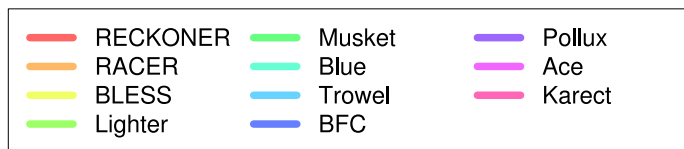
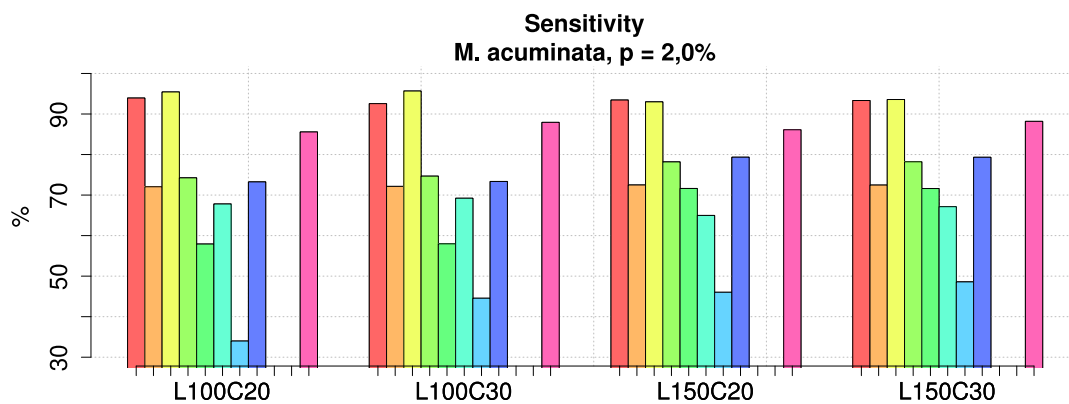
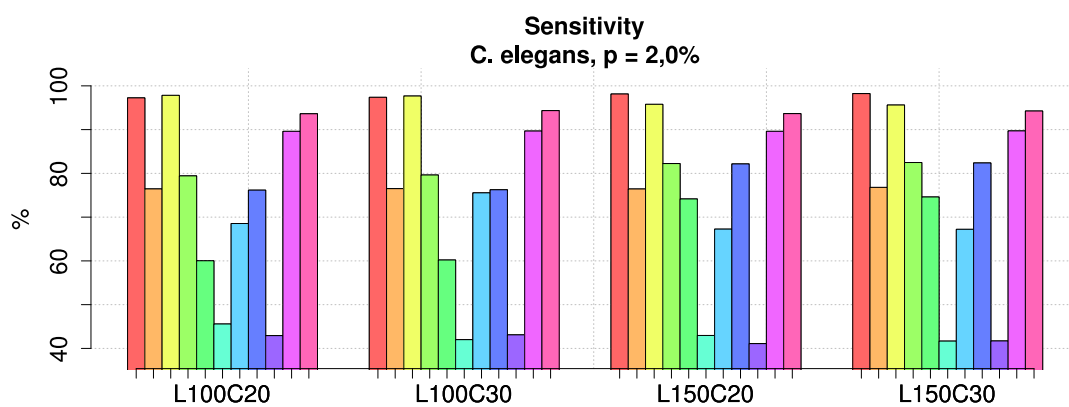
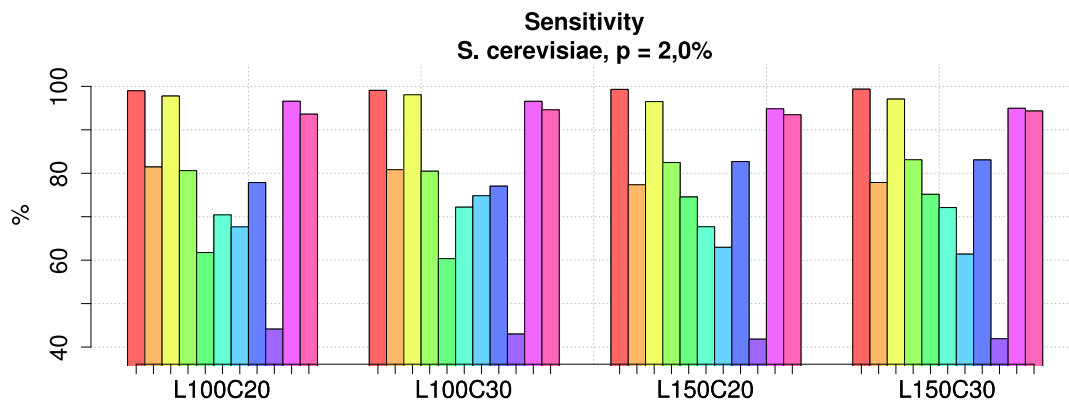
Pattern name	Accession number	Read length	Base error probability
D1	DRR031158	100	2.0 %
D2	SRR065390	100	5.5 %
D3	SRR1802178	150	1.9 %
D4	SRR650760	151	4.2 %

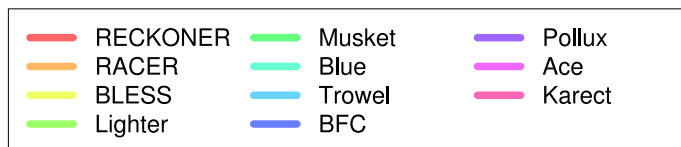
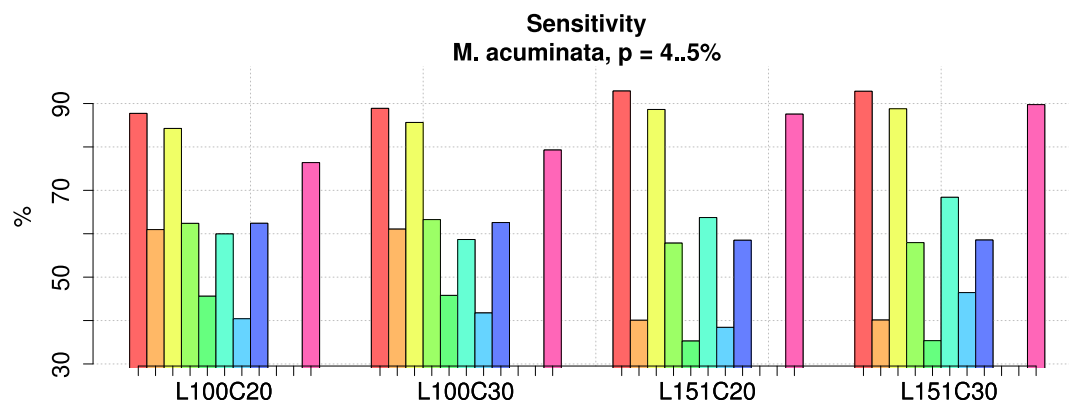
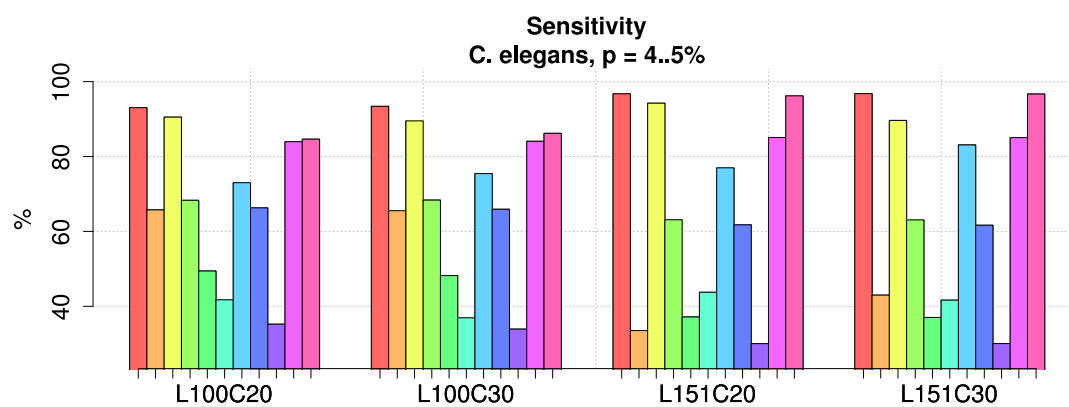
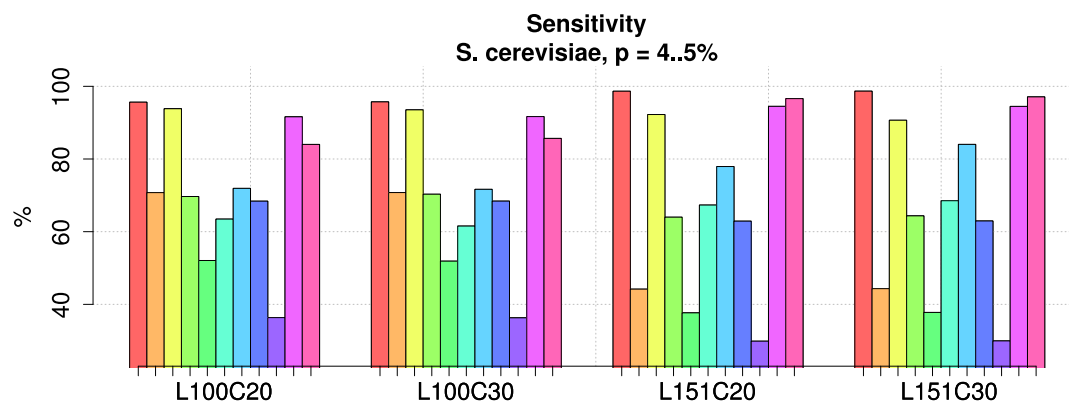
Table 2: Simulated data generation patterns

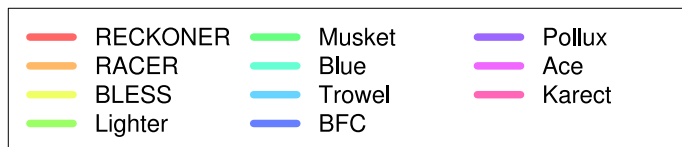
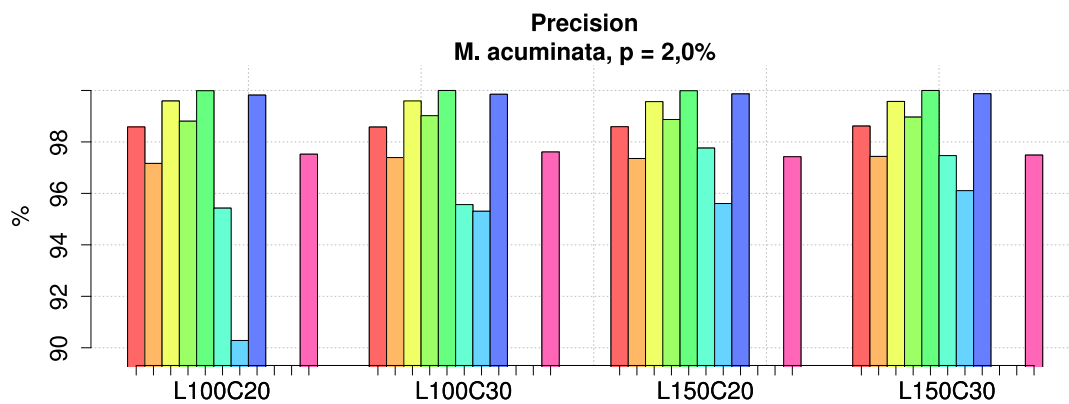
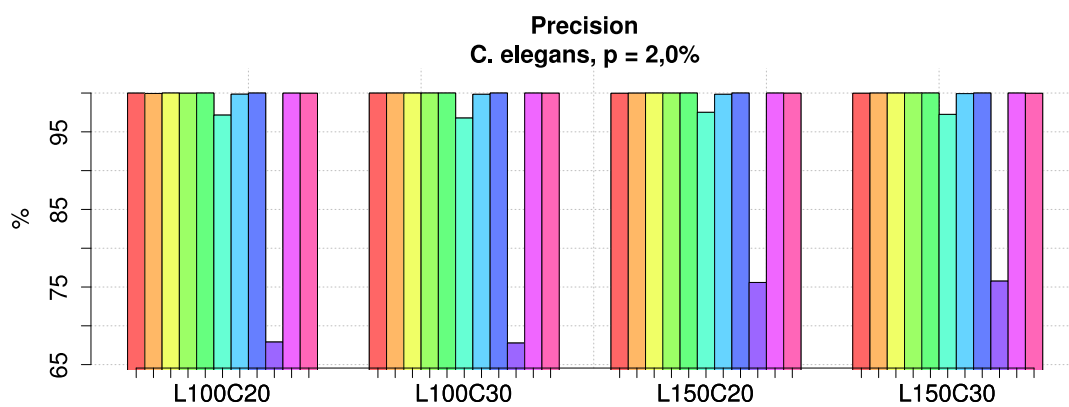
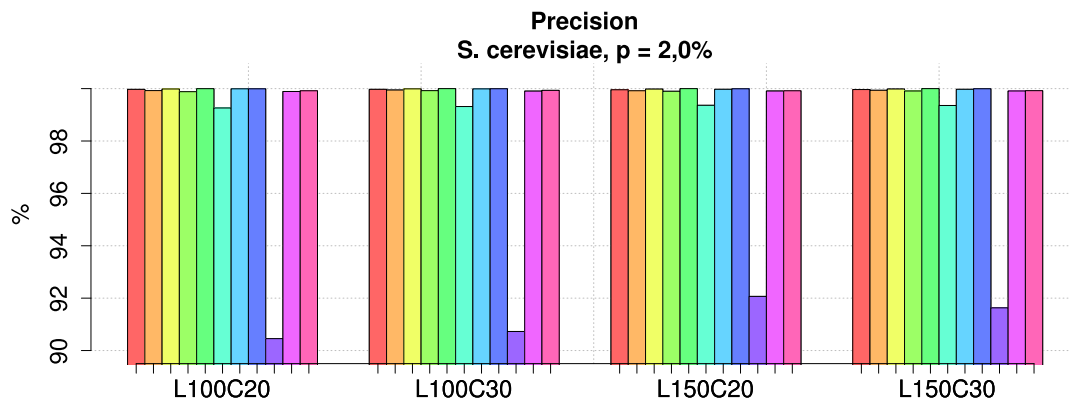
## 2.1 Result quality

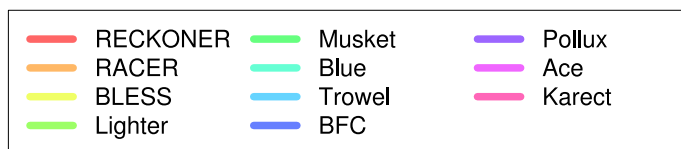
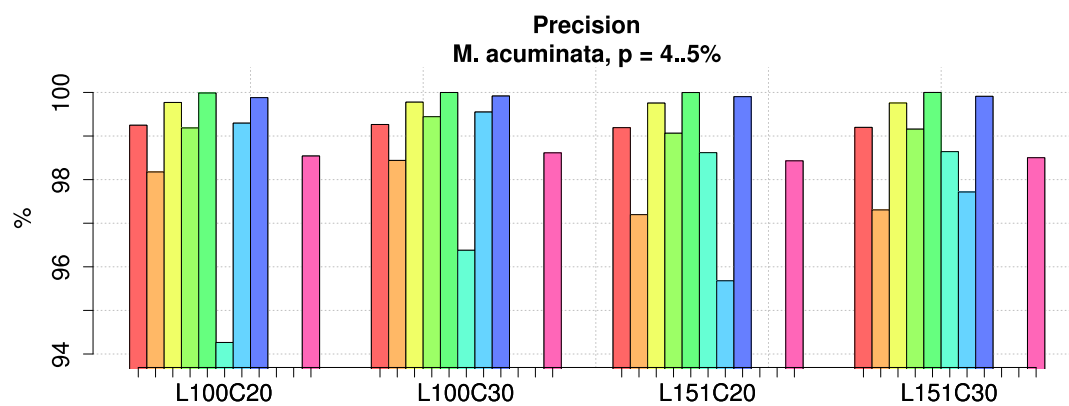
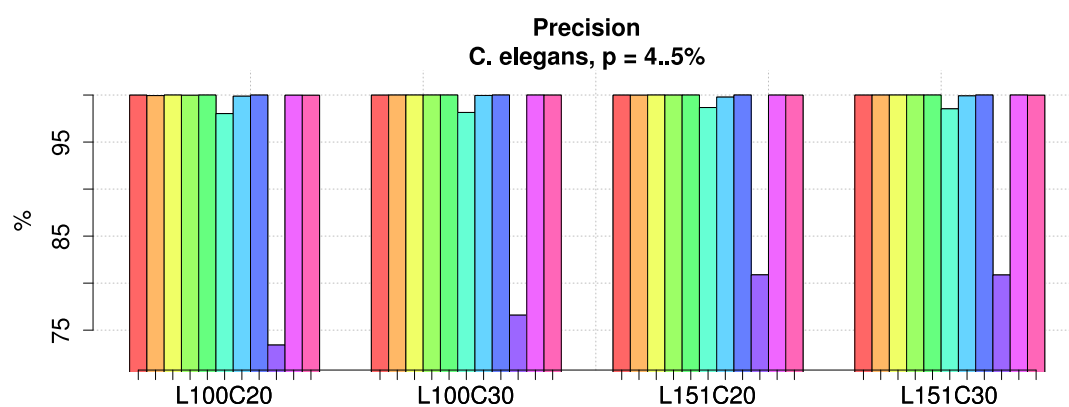
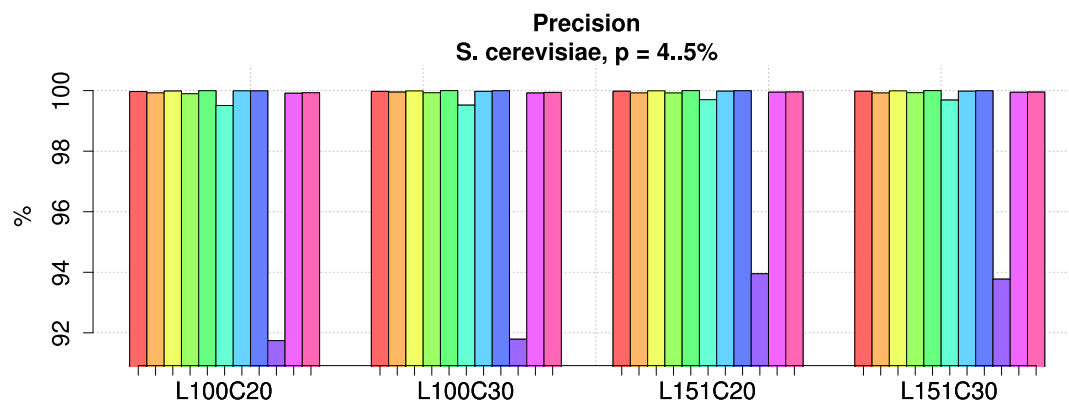






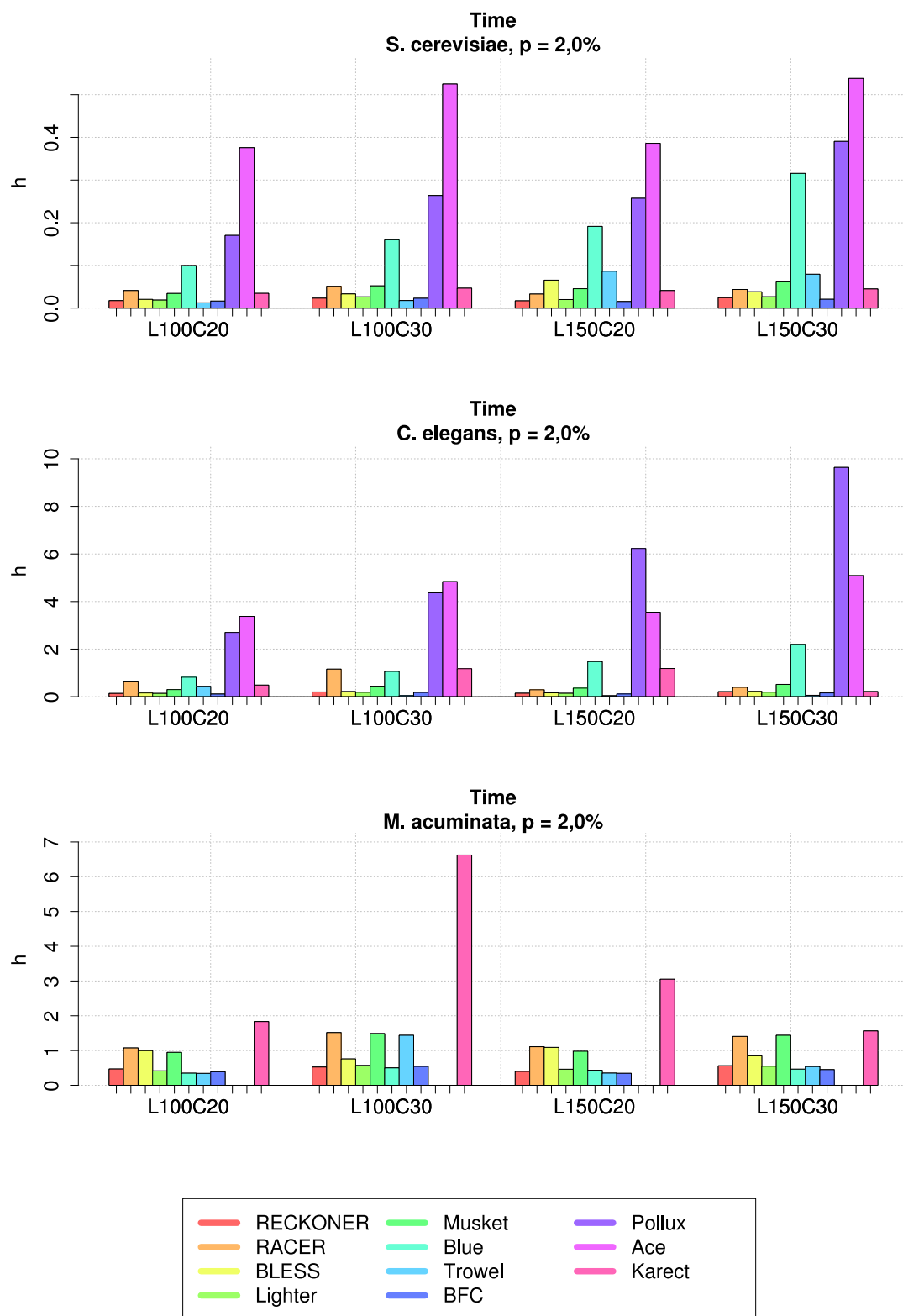


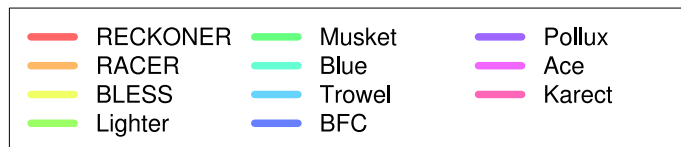
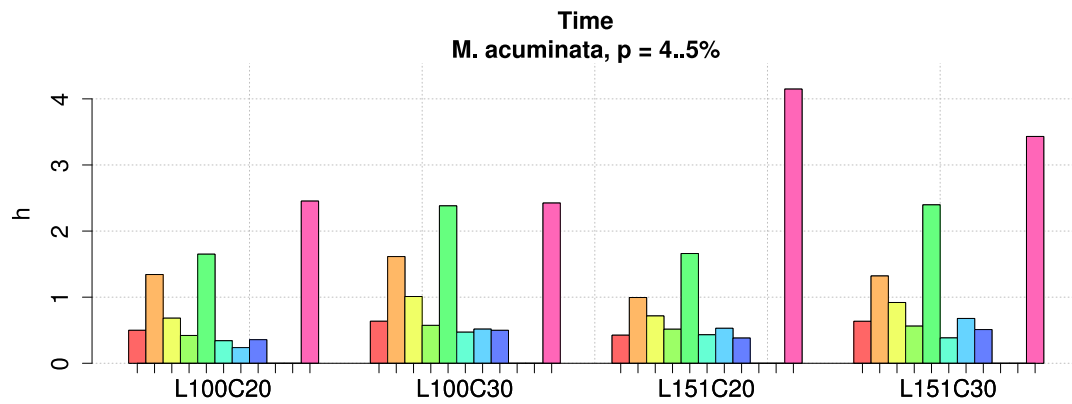
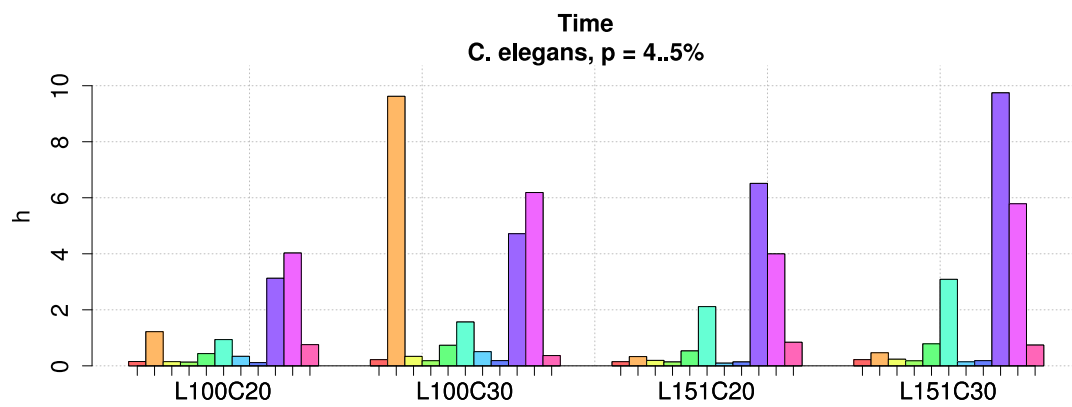
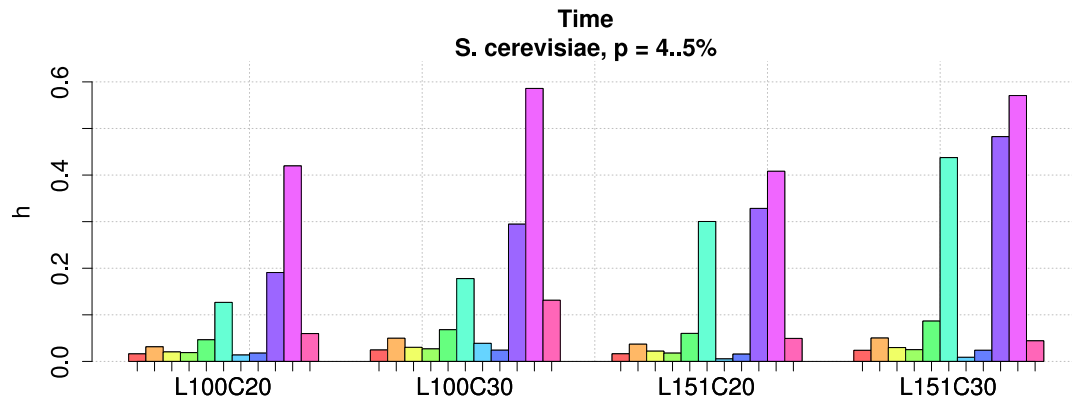


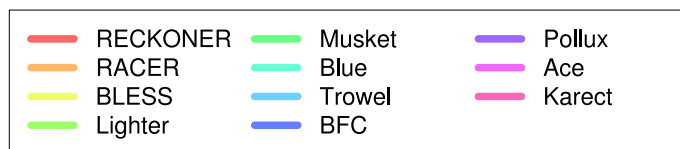
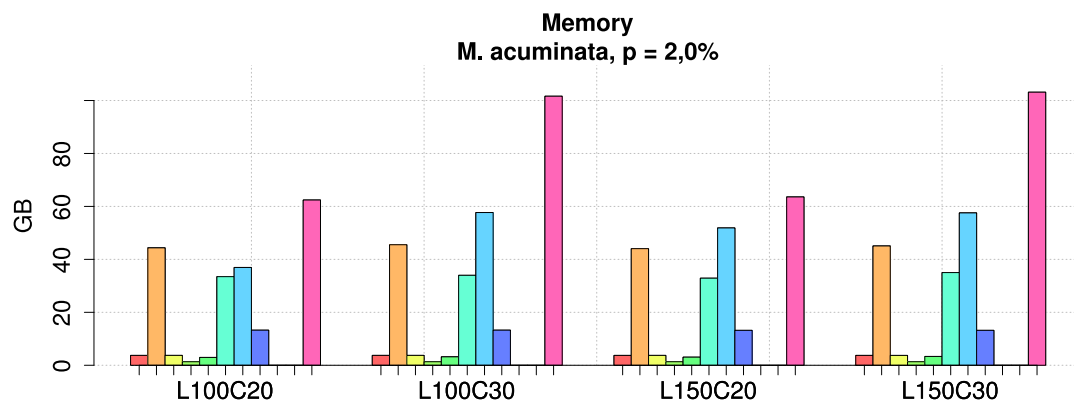
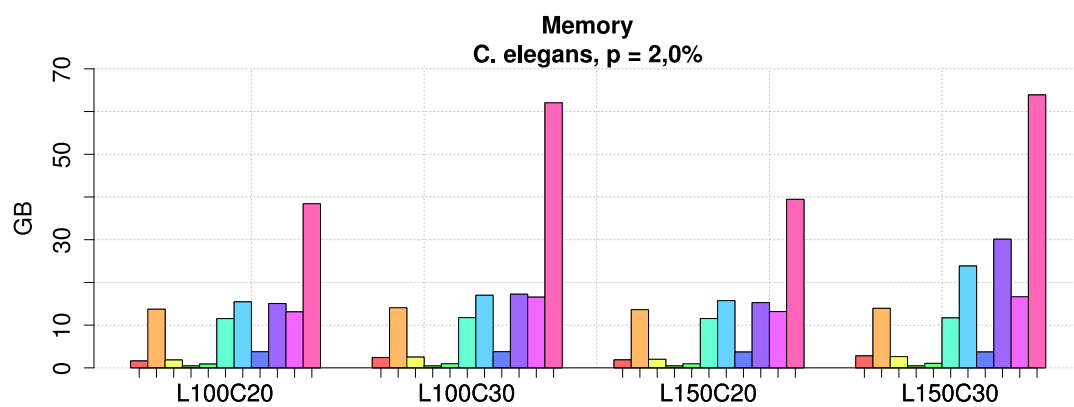
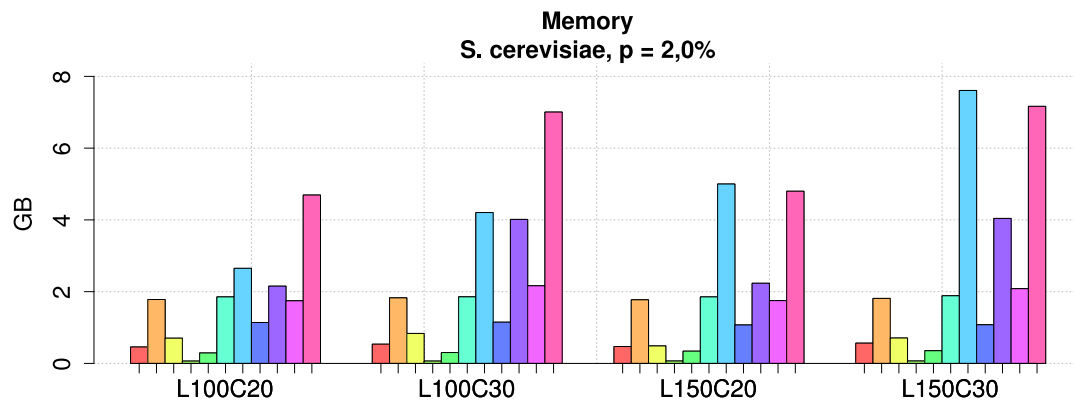


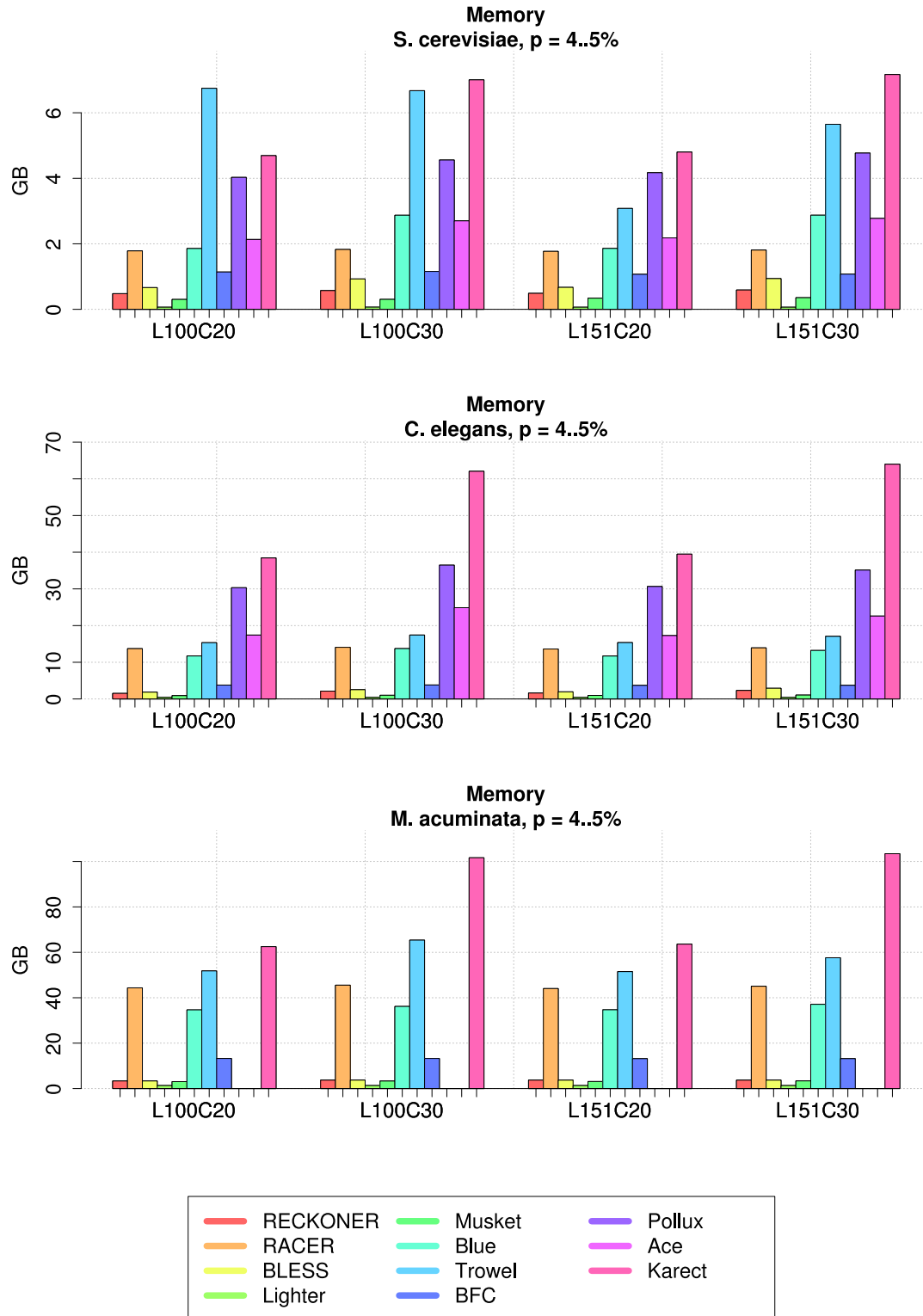


## 2.2 Correction time and memory requirements







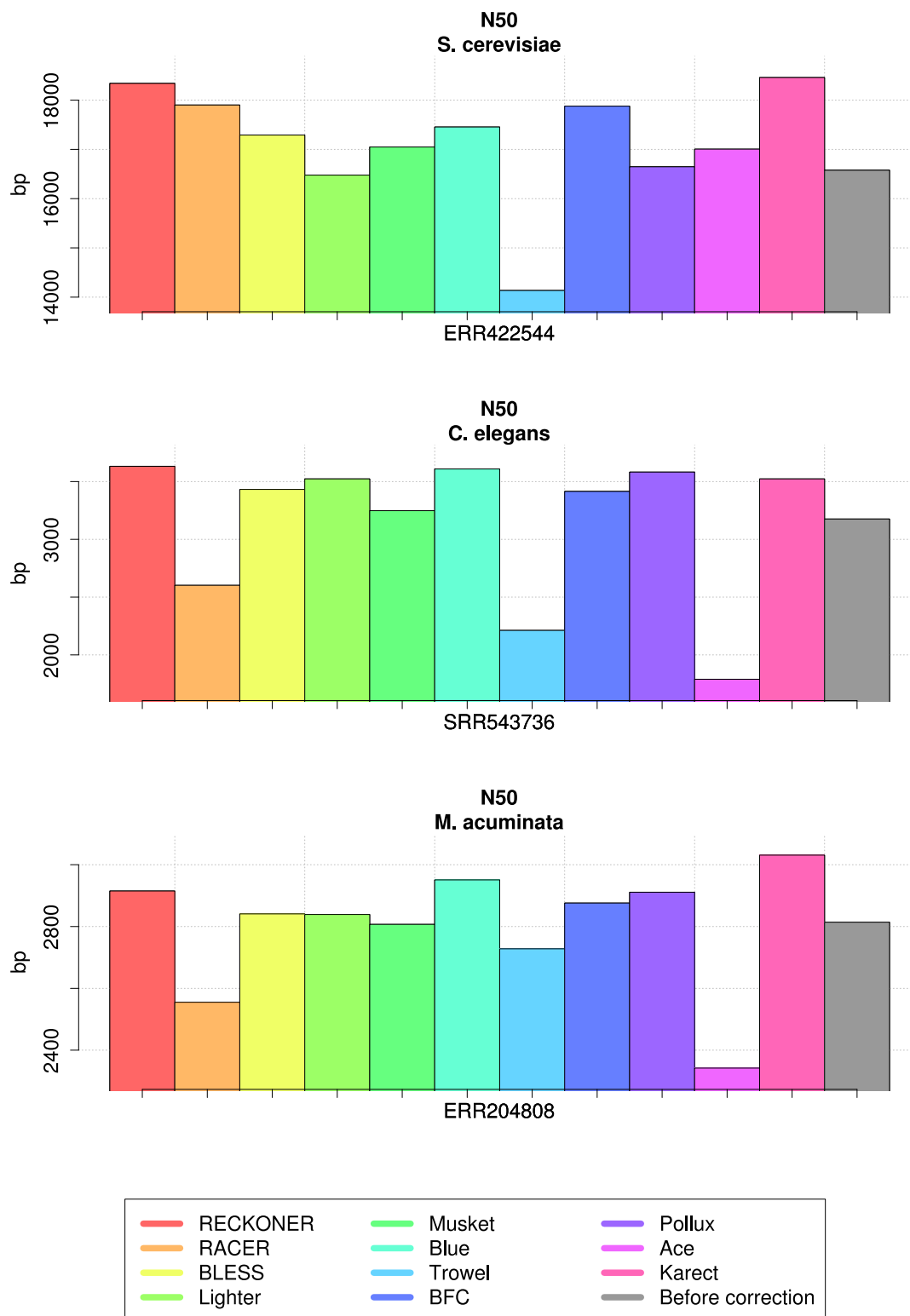


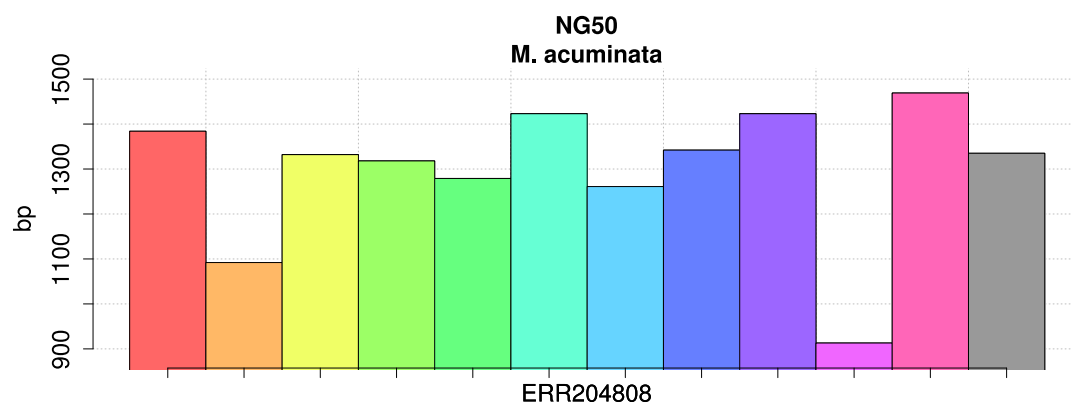
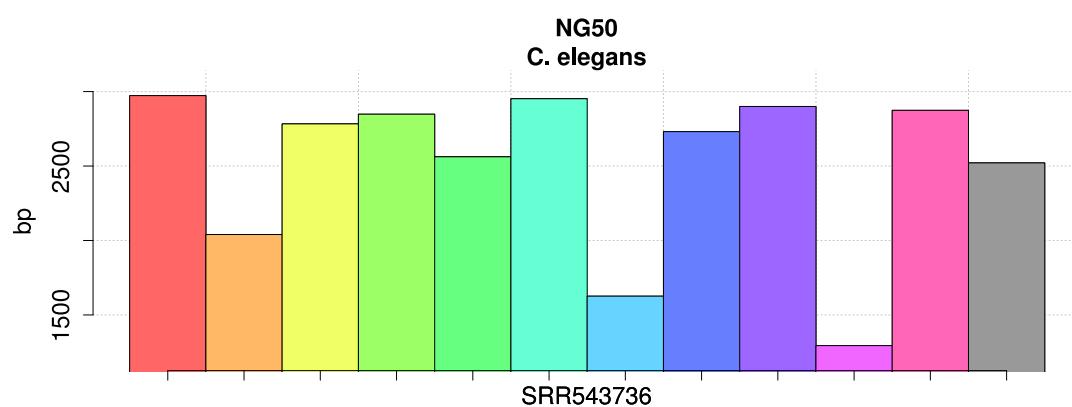
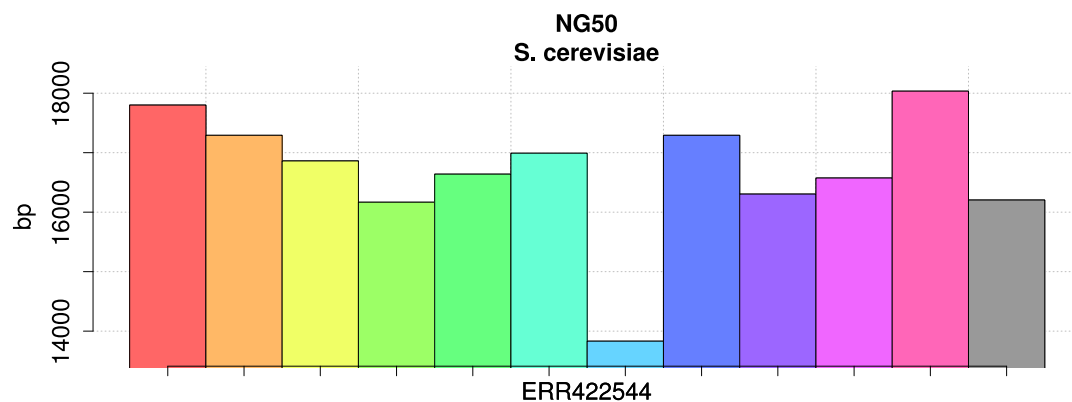
### 3 Results of *de novo* assembly

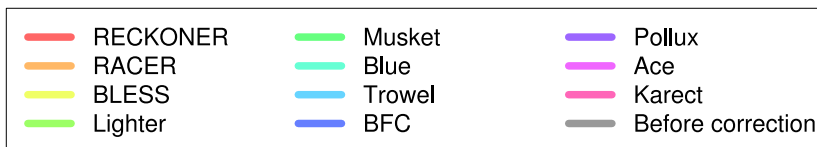
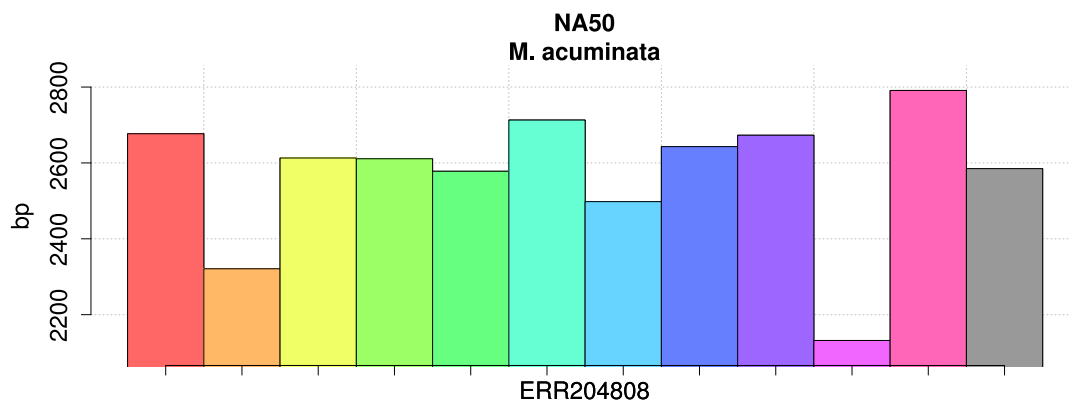
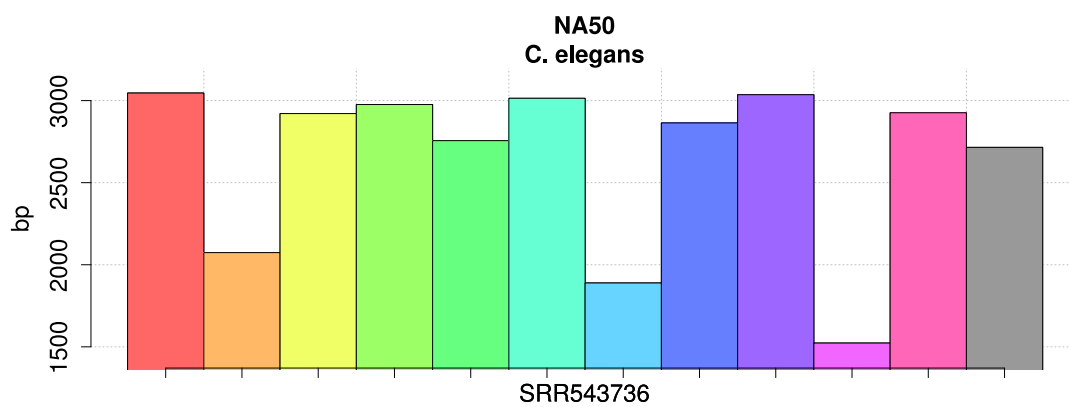
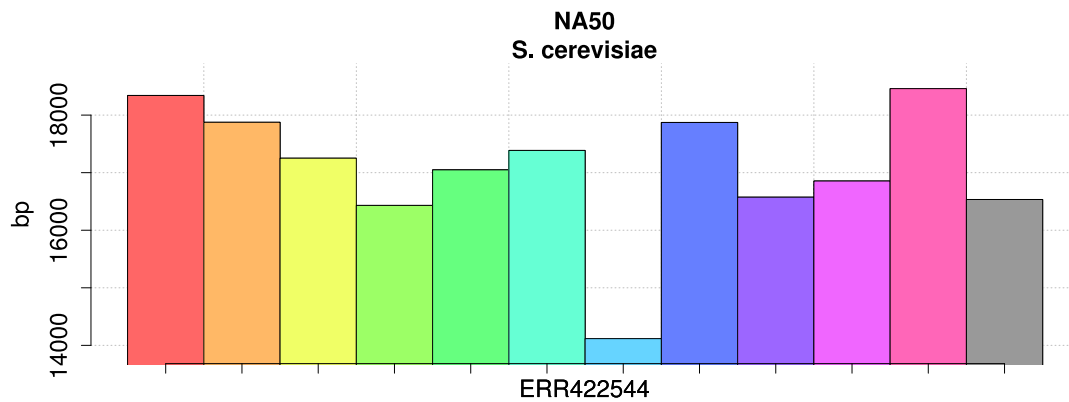
The tests of impact of the correction on *de novo* assembly complies change of assembly quality indicators (N50, NG50, NA50) and the assembler memory and computational time requirements. Missing bars denote

failures of the correction caused by too huge time consumption ( $> 12$  hours).

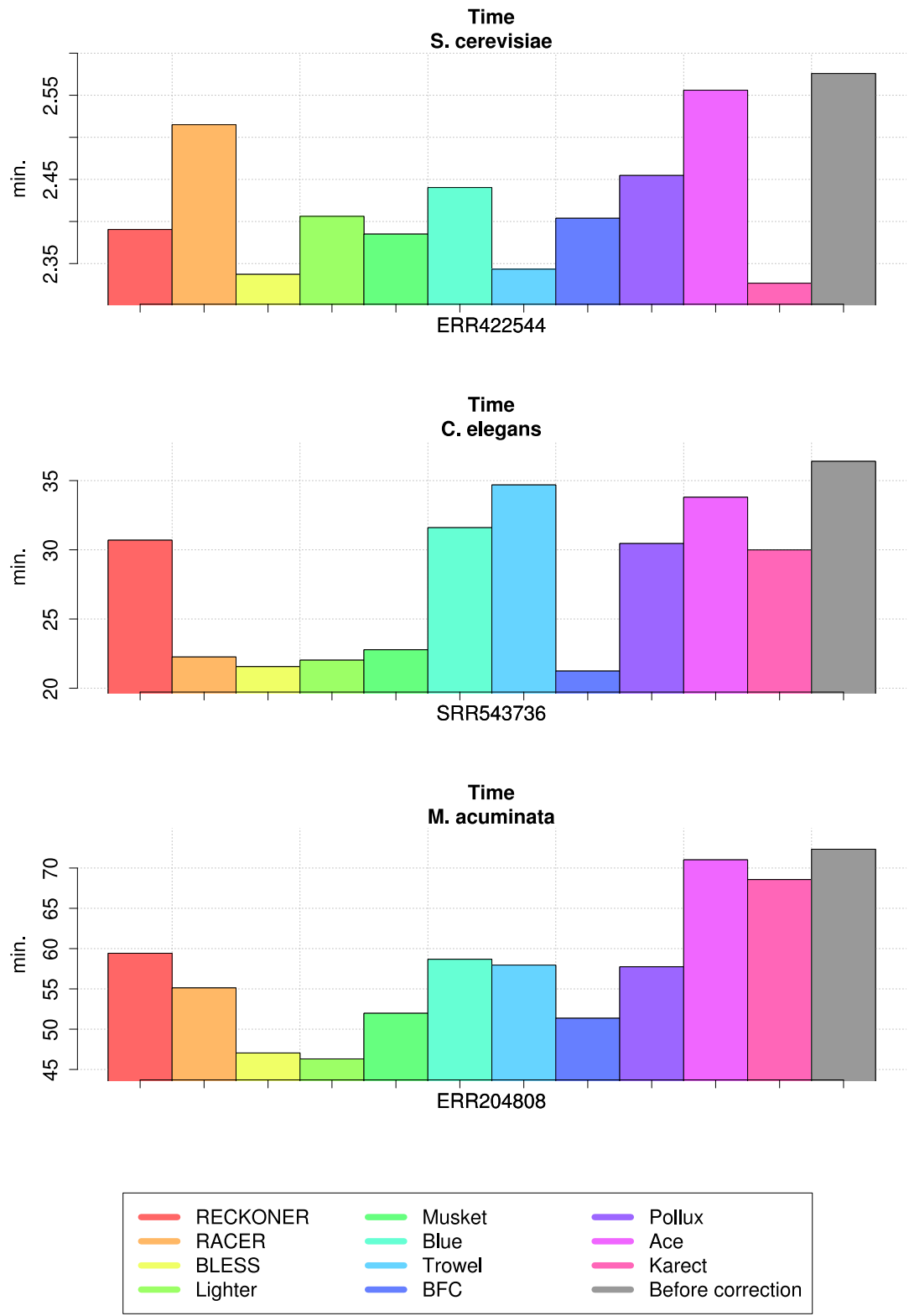
### 3.1 Assembly quality



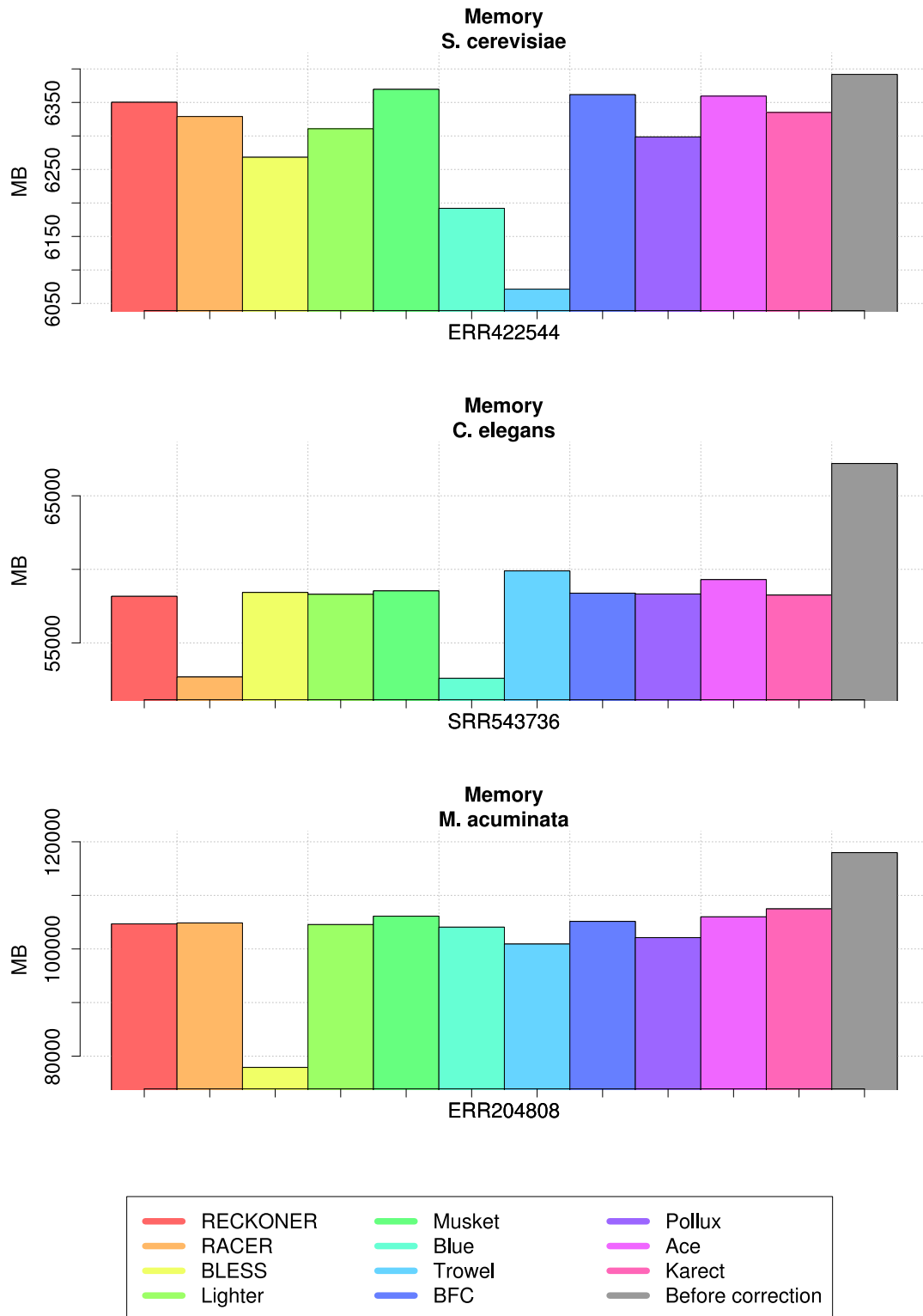




3.2 Assembler time and memory requirements







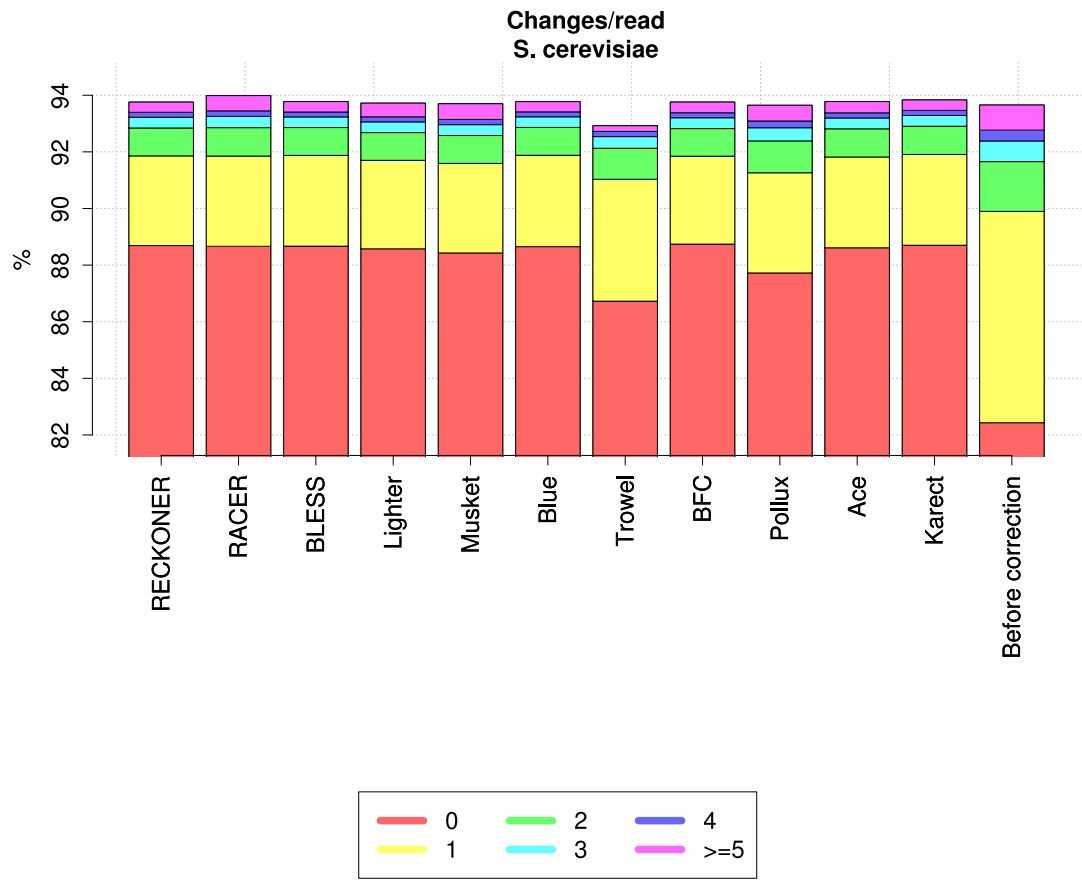
## 4 Results of reads mapping

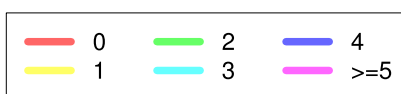
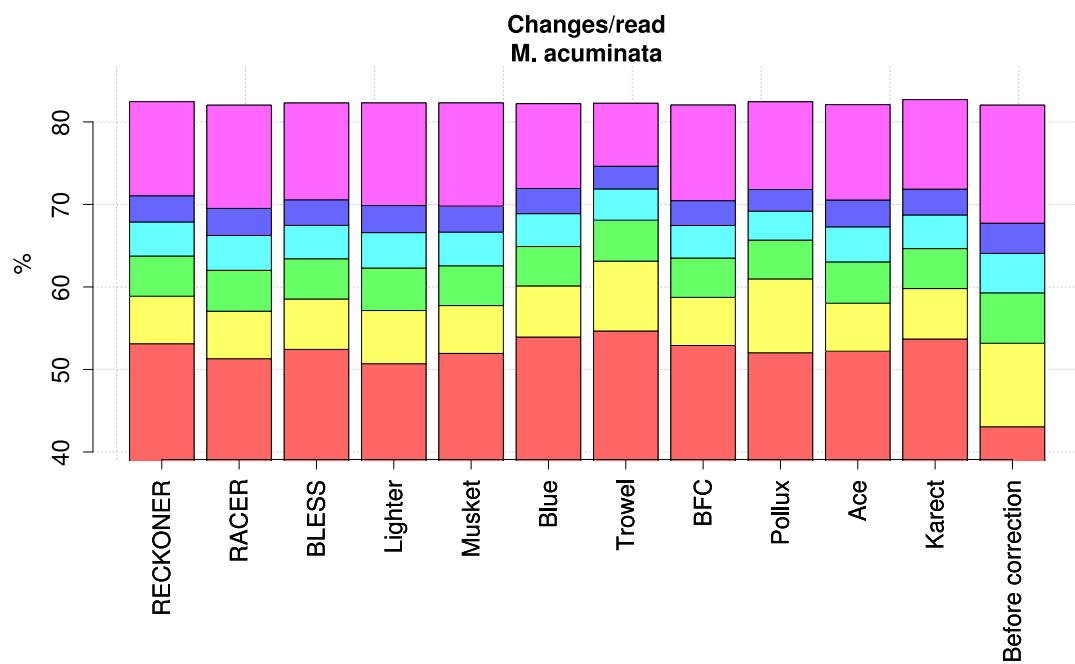
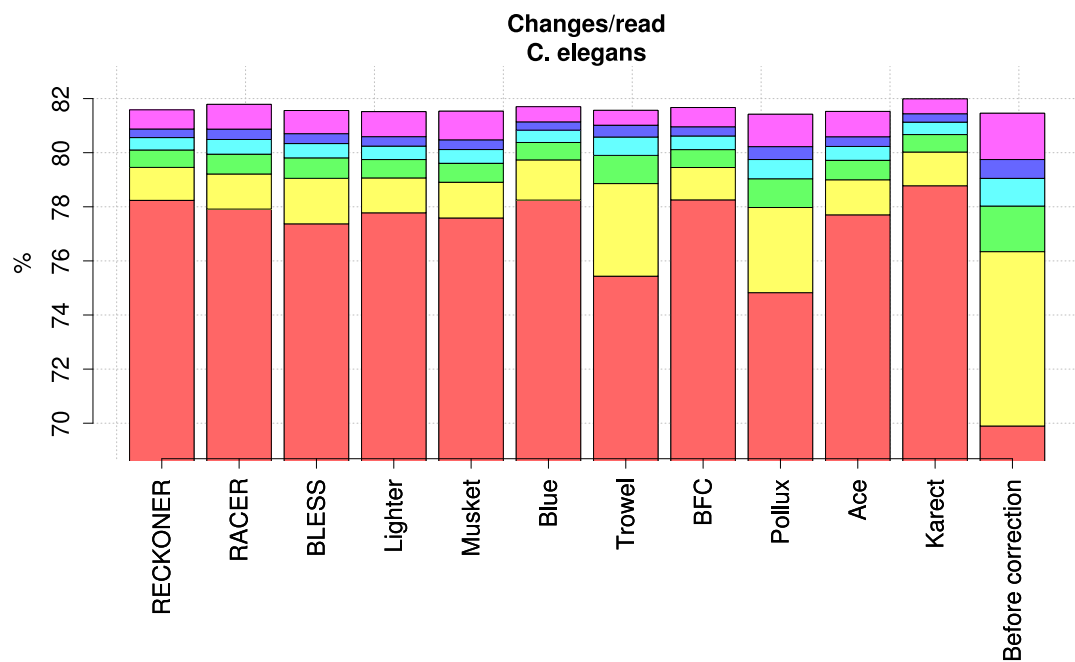
The tests of impact of the correction on reads mapping complies change of number of modifications needed to align the read to a reference genome, change of number of reads aligned once or more than one time to

the genome, and the mapper memory and computational time requirements. Missing bars denote failures of the correction caused by too huge time consumption (> 12 hours).

### 4.1 Mapping quality – number of changes

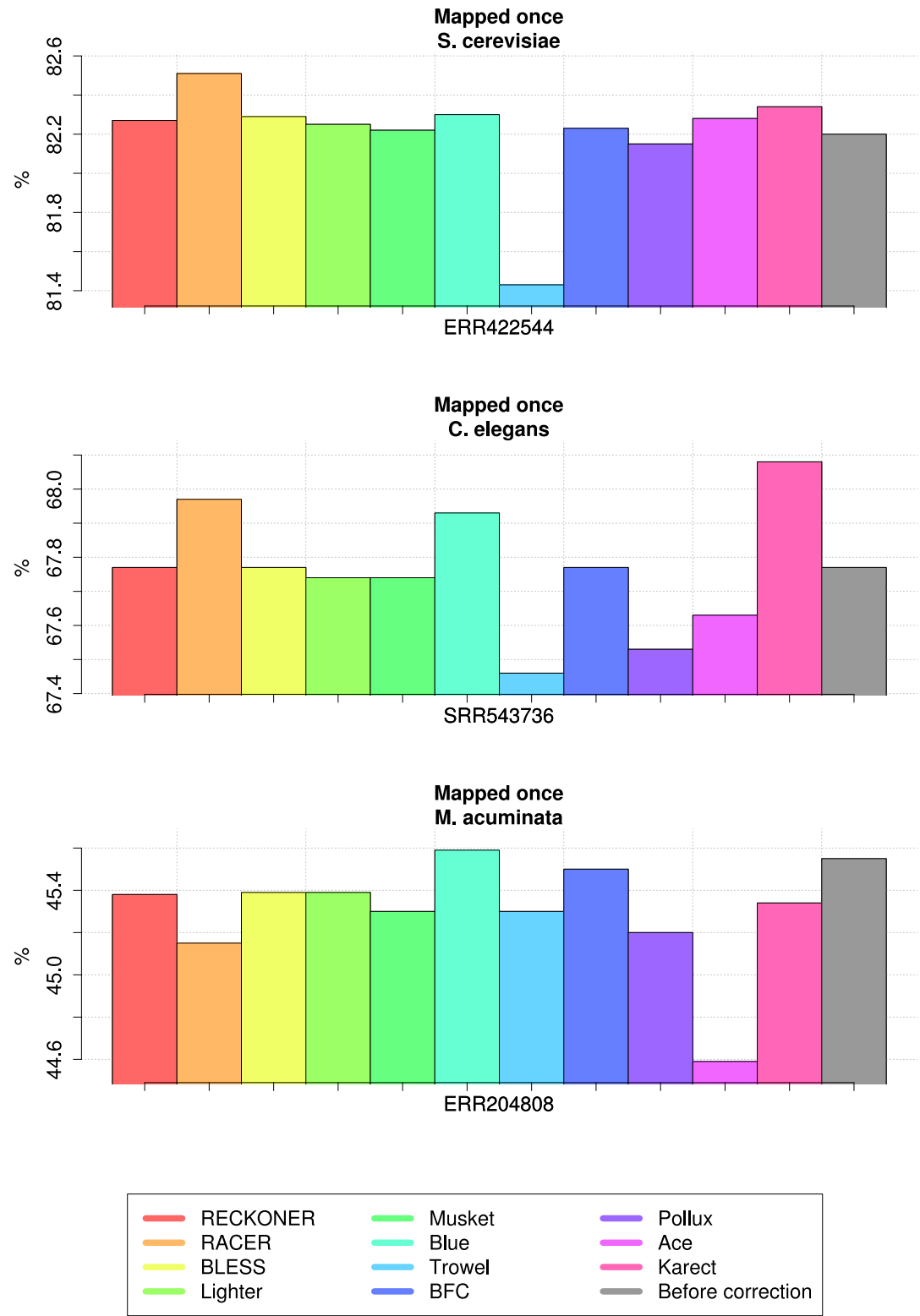
The following plots show number of changes, that had to be introduced to reads before and after the correction in a fraction of reads.

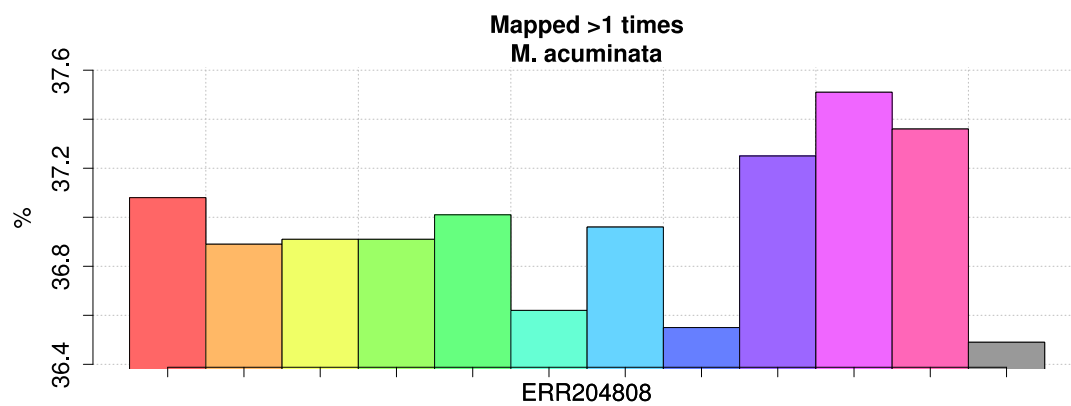
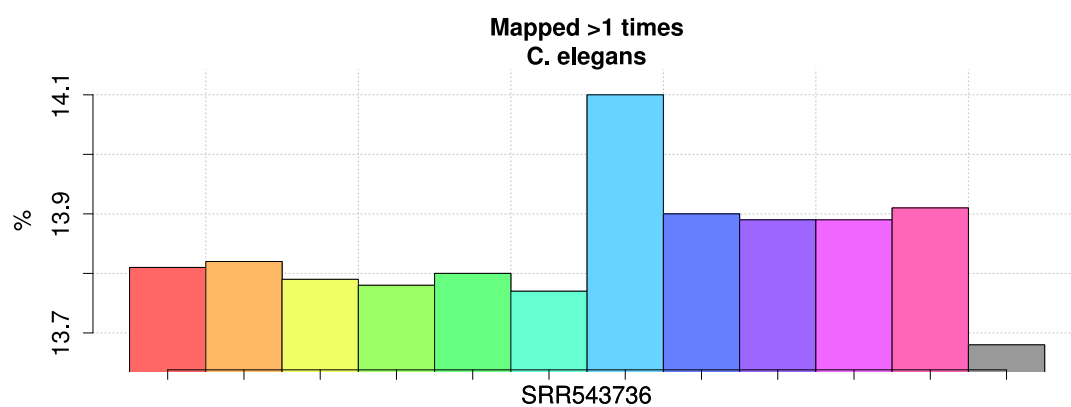
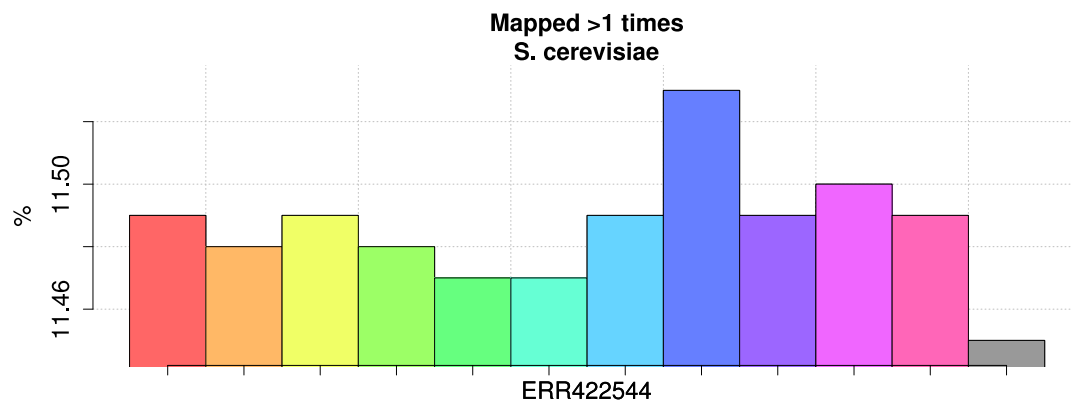




4.2 Mapping quality – number of matches

The following plots show the fraction of reads, that was aligned to the reference genome in one or more places.





4.3 Mapper time and memory requirements

